

AD-A141 576

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM PUBLIC REPORT VOLUME 3(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 25 OCT 83

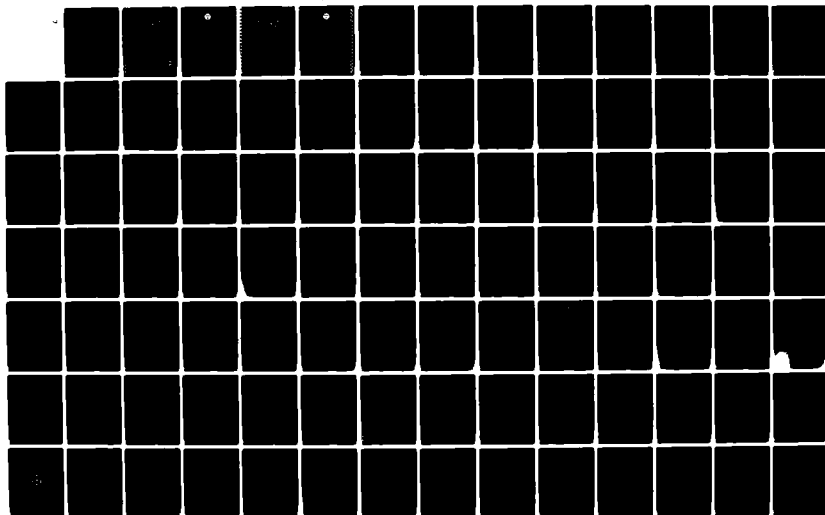
1/5

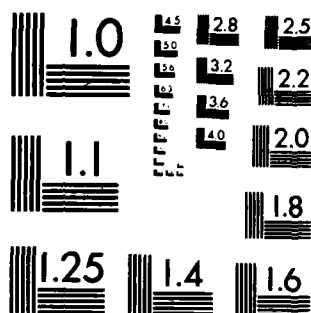
UNCLASSIFIED

NOSC/TD-552-VOL-3

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

①

NOSC TD 552

**KERNEL ADA PROGRAMMING
SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM:
PUBLIC REPORT**

VOLUME III

**Patricia Oberndorf, KIT Chairman
Naval Ocean Systems Center
San Diego, CA 92152**

25 October 1983

Interim Report for 28 October 1982 — 30 June 1983

**Prepared for
ADA JOINT PROGRAM OFFICE
3D139 (400AN) Pentagon
Washington, DC 20301**

**DTIC
SELECTED
MAY 31 1984
S E D**

Approved for public release; distribution unlimited

84 05 30 088

AD-A141 576

DTIC FILE COPY



NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN

Commander

R.M. HILLYER

Technical Director

ADMINISTRATIVE INFORMATION

This report is the third in a series consisting of inputs from the KAPSE Interface Team and its auxiliary industry/academia team. The work was sponsored by the Ada Joint Program Office under program element RDAF, project CS22, sponsor order AF0038AJPO-83-2. The contributions are reproduced here exactly as received.

I would like to extend my appreciation to the many DOD, academic, and commercial activities whose continued support makes this effort possible. The sense of teamwork and cooperation displayed by all members of these two teams is outstanding and will mean the success of what we have undertaken.

Released by
R.A. Wasilausky, Head
C³I Support Systems
Engineering Division

Under authority of
J. Stawiski, Head
Command, Control, Communications
and Intelligence Systems Department

LR

①

NOSC TD 552

**KERNEL ADA PROGRAMMING
SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM:
PUBLIC REPORT**

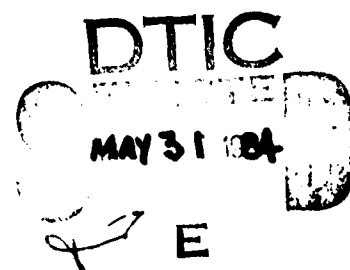
VOLUME III

**Patricia Oberndorf, KIT Chairman
Naval Ocean Systems Center
San Diego, CA 92152**

25 October 1983

Interim Report for 28 October 1982 – 30 June 1983

**Prepared for
ADA JOINT PROGRAM OFFICE
3D139 (400AN) Pentagon
Washington, DC 20301**



Approved for public release; distribution unlimited

84 05 30 088

AD-A141 576

DTIC FILE COPY



NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN

Commander

R.M. HILLYER

Technical Director

ADMINISTRATIVE INFORMATION

This report is the third in a series consisting of inputs from the KAPSE Interface Team and its auxiliary industry/academia team. The work was sponsored by the Ada Joint Program Office under program element RDAF, project CS22, sponsor order AF0038AJPO-83-2. The contributions are reproduced here exactly as received.

I would like to extend my appreciation to the many DOD, academic, and commercial activities whose continued support makes this effort possible. The sense of teamwork and cooperation displayed by all members of these two teams is outstanding and will mean the success of what we have undertaken.

Released by
R.A. Wasilausky, Head
C³I Support Systems
Engineering Division

Under authority of
J. Stawiski, Head
Command, Control, Communications
and Intelligence Systems Department

LR

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 552 (TD 552)	2. GOVT ACCESSION NO. AD-A244 576	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM: PUBLIC REPORT Volume III	5. TYPE OF REPORT & PERIOD COVERED Interim 28 October 1982-30 June 1983	
7. AUTHOR(s) KAPSE Interface Team Patricia A. Oberndorf (NOSC), Chairman	6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152	8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 3D139 (400AN) Pentagon Washington, DC 20301	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS RDAF CS22 AF0038AJPO-83-2	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	12. REPORT DATE 25 October 1983	
	13. NUMBER OF PAGES 300	
	15. SECURITY CLASS. (of this report) Unclassified	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES <i>See p. 1-2</i>		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Computer language KIT Ada KITIA Interface standards KAPSE Programming support systems		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The continuing activities of the Kernel Ada Programming Support Environments (KAPSE) interface team and its industry/academia auxiliary are reported. (Ada is a recent, DOD-developed programming language.) The Ada Joint Program Office (AJPO)-sponsored effort will ensure the interoperability and transportability of tools and data bases among different KAPSE implementations. The effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous ADA-related developments, the widest possible participation is being encouraged to create a broad base of experience and acceptance in industry, academia, and the DOD.		

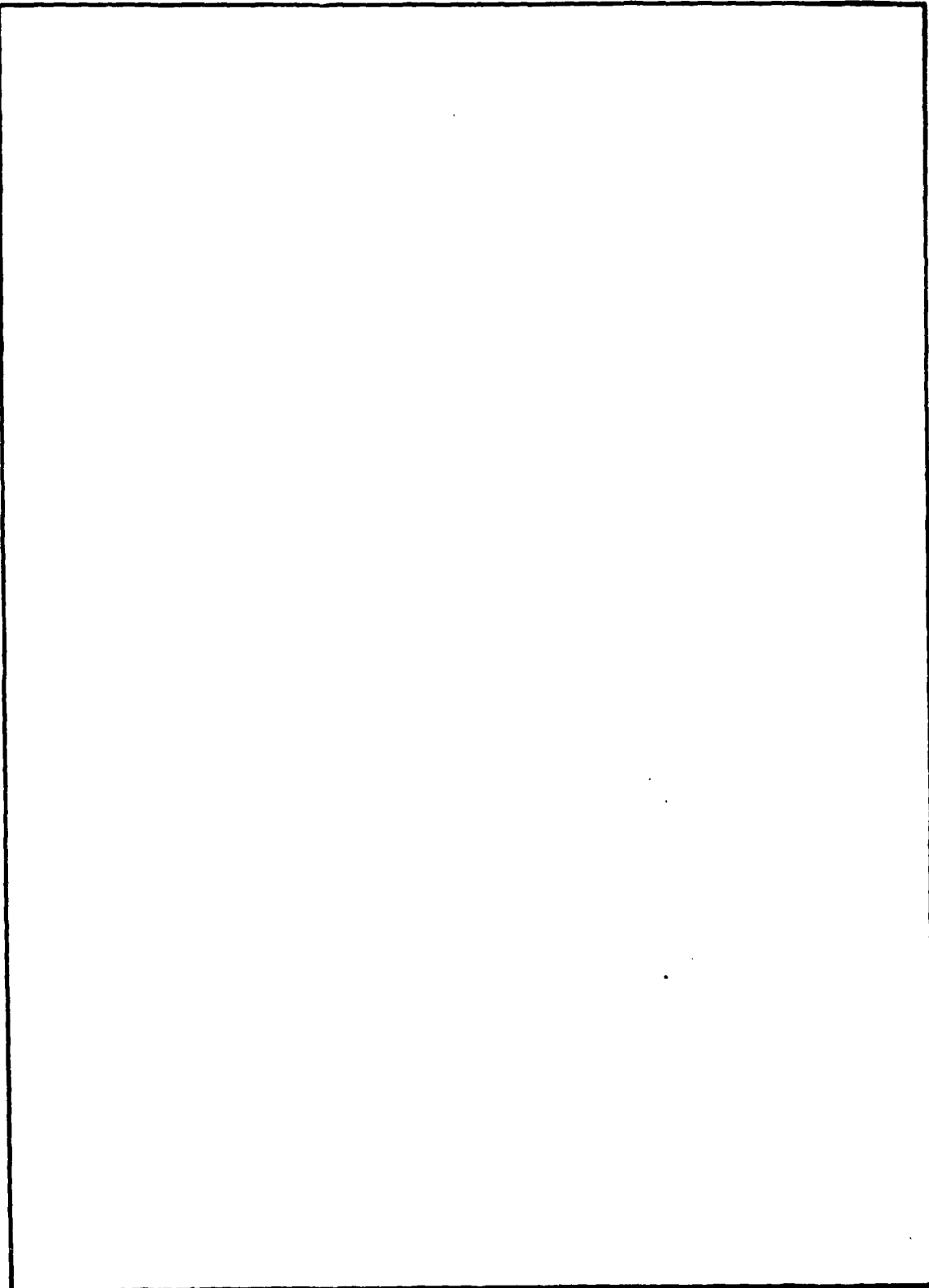
DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102- LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



S/N 0102- LF- 014- 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

1.	INTRODUCTION	1-1
2.	TEAM PROCEEDINGS	
	A. KIT Minutes 25-27 January 1983	2A-1
	B. KITIA Minutes 21-23 February 1983	2B-1
	C. KIT/KITIA Minutes 18-21 April 1983	2C-1
	D. SIS Drafter's Minutes 8-9 March 1983	2D-1
3.	KIT/KITIA DOCUMENTATION	
	A. Ada Programming Support Environment Interoperability and Transportability (I&T) Management Plan	3A-1
	B. APSE Interoperability and Transportability Implementation Strategy	3B-1
	C. KAPSE File Structure	3C-1
	D. A Virtual Terminal Specification and Rationale	3D-1
	E. Program Invocation and Control	3E-1
	F. SIS Implementation Issues: Parameter Passing Over the Standard Interface	3F-1
	G. SIS Categories	3G-1
	H. KAPSE Support for Program/Terminal Interaction	3H-1
	I. The Difficulty in Developing an Ada Environment for Both Run-Time and Programming Support Environments	3I-1
	J. Minimal Host for the KAPSE	3J-1
	K. Of Mice and Command Languages: KAPSE Interface Support for Interactive Tools	3K-1
	L. Evolution of an APSE Interface Tool	3L-1
	APPENDIX A. Configuration Management System Interim Report on Interface Analysis	A-1
	APPENDIX B. APSE Interactive Monitor Interim Report on Interface Analysis and Software Engineering Techniques	B-1
	APPENDIX C. Ada-Europe/AdATEC Conference Briefing	C-1

SECTION 1

INTRODUCTION

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Avail and/or Special
A-1	



INTRODUCTION

This report is the third in a series that is being published by the KAPSE Interface Team (KIT). The first was published as a Naval Ocean Systems Center (NOSC) Report, TD-509, dated April 1982, and is now available through the National Technical Information Service (NTIS)* for \$19.50 hardcopy or \$4.00 microfiche; ask for order number AD A115 590. The second was published as NOSC TD-552, dated October 1982, and is now available through NTIS for \$44.50 hardcopy; ask for order number AD A123 136.

This series of reports serves to record the activities which have taken place to date and to submit for public review the products that have resulted. The reports are issued approximately every six months. They should be viewed as snapshots of the progress of the KIT and its companion team, the KAPSE Interface Team from Industry and Academia (KITIA); everything that is ready for public review at a given time is included. These reports represent evolving ideas, so the contents should not be taken as fixed or final.

MEETINGS

Both teams have held two additional meetings since the last report: a KIT meeting in January, 1983, a KITIA meeting in February, 1983, and a joint meeting of the two teams in April, 1983. The approved minutes from the January and February meetings are included in this report. In addition, many of the working groups held individual meetings between regular KIT/KITIA meetings.

THE APSE INTEROPERABILITY AND TRANSPORTABILITY (I&T) PLAN

A new version of this plan is included in this report. It reflects some minor changes in thinking, but the most significant changes are in the schedules. This version of the plan also includes a more detailed and complete

*National Technical Information Service (NTIS)
Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

work-breakdown structure, which clearly states the things the KIT and KITIA plan to accomplish during the next three years.

THE STANDARD INTERFACE SET (SIS)

In the last report a section was included which discussed the APSE Integrated Environment (AIE)/Ada Language System (ALS) Analysis. It mentioned meetings which were taking place between KIT members and representatives of Intermetrics and SofTech to construct a set of interfaces on which the AIE and ALS could agree. These meetings were expanded in February to include representatives of the KITIA as well, and several more meetings have taken place. Minutes from the March Standard Interface Set Working Group (SISWG) meeting are included in this report. The result has been a set of steadily maturing interfaces which were partially presented at the April KIT/KITIA meeting and which will be presented in full at the July meeting. Several more meetings are planned following the July presentation to ready these interfaces for presentation at a public review scheduled by the Ada Joint Program Office (AJPO) for next fall.

The current version of the interface set has not been included in this report because of its lack of maturity and lack of approval by the KIT and KITIA. It is important to note that this set is still based solely on those interfaces which both the AIE and ALS could support without changes to their existing designs. They have not been examined for completeness or consistency; nor have they been examined with respect to fulfillment of the I&T Requirements and Criteria, which are also being drafted at this time (see next section of this Introduction). The formulation of this initial interface set serves several purposes:

- 1) It clarifies the similarities and differences between the ALS and AIE. Such knowledge will help the KIT and KITIA to establish a final interface set that does not differ arbitrarily from the capabilities which the AIE and ALS can provide.
- 2) It provides the KIT and KITIA with a greater understanding of some of the interface issues which must be addressed in establishing an SIS.

3) It raises some new interface issues which the teams were unaware of at the beginning of the project.

4) It gives the public some early insight into what kind of product is to be expected of the KIT/KITIA effort.

5) It provides a starting point for the evolution of a proper set of interfaces for achieving APSE I&T.

The version presented at the review in the fall will be included in the next KIT public report.

REQUIREMENTS AND CRITERIA

Considerable progress was made in the formulation of I&T Requirements and SIS Design Criteria. The document was significantly reorganized due to clarification of what constitutes a requirement as opposed to a criterion. Both teams have devoted more time to reviewing the document than before, and consensus on many issues is forming. As with the initial version of the SIS, the current document has not been included in this report due to its state of flux. It is expected that a final version will be available to the public by the end of this calendar year. The final document will serve as the motivation for development of a SIS.

POLICY DISCUSSIONS

The policy discussions prompted by the KITIA and expanded upon by the KIT have culminated in the completion of the document described in the last public report. Called the APSE I&T Implementation Strategy, it discusses various issue areas, the considerations and trade-offs associated with them, and the choices which the KIT have made after considering the alternatives. This document provides an indication of the KIT approach which complements the I&T plan and clarifies the reasons for some of the I&T plan contents. This report is included here in Section 3.

KIT AND KITIA PAPERS

Two working papers by members of the KIT are included in Section 3. The authors are Krutar, Naval Research Lab (NRL), and French, Texas Instruments (TI). Four papers authored by KITIA members also appear in this section. They include an updated version of a paper by Gargaro, Computer Sciences Corporation (CSC), which appeared in the last Public Report, and the slides which Gargaro presented on KIT/KITIA activities at the joint conference of Ada-Europe and AdaTec in Brussels in March of this year.

I&T TOOLS

Two tools are being developed to assist the KIT in discovering and evaluating interface issues. The first of these two is Configure, under development by the Falls Church, Virginia, office of CSC. The second is the APSE Interactive Monitor (AIM), under development by the Lewisville, Texas, office of TI. Both of these tools have proceeded to detailed design; a critical design review (CDR) was held with CSC in June and one with TI is scheduled for July.

At this juncture there is no plan to proceed with the implementation of the Configure tool. This decision has been prompted by two things. One is a lack of understanding of what is required for configuration management (CM) in the context of Ada and APSEs. The second is the fact that, while Configure is a good tool concept, its generality has resulted in very little substantial insight into AIE and ALS interfaces. It was originally hoped that such a tool would delve very deeply into data base issues, but that has not been the case. The CSC Interface Analysis report is included in this volume as Appendix A.

In an attempt to answer the first problem — the lack of understanding of Configuration Management (CM) for Ada and APSEs — a configuration management workshop was convened in June. A small group of knowledgeable people from a wide variety of backgrounds was assembled to take up the question. The people ranged from Ada experts to CM experts who knew little about Ada, and the group included two representatives of APSE development efforts. The most interesting conclusion of the workshop attendees was that Ada does not radically

change configuration management for large Department of Defense (DoD) systems. Although it does make some contributions toward solving the CM problem which languages in the past have not directly provided, Ada makes no new demands on CM and does not solve the problem alone. The report from this workshop will be included in the next public report.

The AIM development has provided considerable insight into APSE interfaces, as reflected in the report which is included here as Appendix B. Work in this area has also contributed substantially to developing the initial SIS, and a representative of the AIM team has attended recent meetings of the SIS working group.

OTHER KIT/KITIA ACTIVITIES

A public review of several of the latest B-5 specifications for the AIE has been conducted over the last few months. The analysis report of this review will be included in the next Public Report.

CONCLUSION

This Public Report is provided by the KIT and KITIA to solicit comments and feedback from those who do not regularly participate on either of the teams. Comments on this and all subsequent reports are encouraged. They should be addressed to:

Patricia Oberndorf
Code 8322
Naval Ocean Systems Center
San Diego, CA 92152

or sent via ARPANET to POBERNDORF@ECLB.

SECTION 2

TEAM PROCEEDINGS

KIT MINUTES
MEETING OF 25 - 27 JANUARY 1983
TRW LSI PRODUCTS DIVISION
SAN DIEGO, CALIFORNIA

ATTENDEES : SEE APPENDIX A

BIBLIOGRAPHY OF HANDOUTS : SEE APPENDIX B

25 JANUARY 1983

1. OPENING REMARKS/GENERAL NEWS

Tricia Oberndorf, the KIT chairperson, brought the meeting to order and welcomed everyone to San Diego. Highlights follow;

- . Hospitality packages are available for those members that are not familiar with the San Diego area.
- . A dinner and meeting will be held tomorrow night for LCDR Jack Kramer in honor of his retirement from the Navy.
- . The national AdaTEC meeting will be held in San Diego during the month of February.
- . The KITIA will meet the same week, members must get clearance to attend from Edgar Sibley the KITIA chairperson.
- . A volunteer is needed to attend the International AdaTEC meeting which will be held in Brussels during the month of March.
- . A status report of the Software Technology Initiative (STI) meeting to be held in February was given.
- . The relationship of the KIT and the STI will be discussed at a later date.
- . The Public Report is being distributed. Members who have not yet received theirs should receive them shortly.
- . Due to the anticipated size of the next Public Report, instructions for publication will be distributed for submitters.
- . Documents for the AIE review are being released this week. A schedule for comments is being proposed.
- . Examples of the logo were passed out. Several problems were cited with them. Ideas and comments were solicited.
- . The October minutes will be passed out later for review.

- . Proposed dates for the upcoming KIT meeting are April 19 - 21 in Warminster, PA. and sometime in July for the San Diego meeting.

2. KIT/KITIA ACTIVITIES

Ideas were exchanged as to when and how the KIT and KITIA could meet both jointly and independently. A new approach to the interactions of the KIT/KITIA for joint productivity of the I&T effort was proposed. Other topics in the discussions included:

- . A need was expressed for a clear understanding of what the KITIA's input to the KIT should be.
- . A clearer definition of the KIT's and the KITIA's products and roles was called for.

ACTION ITEM: Tricia will create and the KIT Executive committee will review a sample statement of roles for the KIT/KITIA. The document will be put on the net for review and comment. It will be fashioned after the APSE I&T Plan which is included in the Public Report II.

- . The differences between the two groups were explained, each groups' mode of operation was outlined.
- . Tricia requested the KIT working group chairpersons to open communication lines with their corresponding KITIA counterparts.
- . The KIT and KITIA address lists were updated and placed in KIT-INFORMATION.
- . AJPO funding for KIT traveling expenses was discussed.
- . The role of the STI and the KIT, its products and their operation were explained.

A vote was taken to resolve if the group wanted to meet with the KITIA. It was decided that the KIT should meet with the KITIA at least once a year. No meeting place was proposed although a joint meeting with AdaTEC was suggested.

3. GENERAL NEWS

General news was discussed and included;

- . the distribution of letters of appreciation
- . status of CONFIGURE, CSC's configuration management tool
- . ideas on revision to STONEMAN passed out for review and follow-up presentation
- . status of AIM, TI's interactive monitor tool - PDR on Friday

- . RFP for new tool due to be released soon
- . document containing initial set of interfaces that can be supported by the AIE & ALS systems passed out
- . paper by W. Wilder (SofTech) discussing the parameters for a minimum host passed out
- . considerations for potential legal problems pertaining to the KIT and the Ada effort were discussed

4. BREAK FOR LUNCH

5. RECONVENE

6. REQUIREMENTS AND CRITERIA DOCUMENT

A review of sections 4, 5 & 6 of the Requirements & Criteria document was performed by the KIT. The group concentrated on responding to issues and statement contents. Language ambiguities were corrected only when obviously incorrect. The following topics were discussed:

- . the changing of the acronym "IT" to "I&T" for the abbreviation of Interoperability and Transportability
- . the abolishment of the acronym S_I_S
- . a separation of each section was proposed where all requirements are grouped together followed by all the criteria for each section
- . the document should include only positive specifications
- . considerations for adaptation of another type of organization similar to STONEMAN
- . the ingredients of transportability among tools and among inter-tools that the KIT is attempting to achieve
- . a glossary of standard definitions to be developed in a separate document
- . the definitions of design criteria and requirements
- . considerations for making something a requirement
- . policy issues such as upward compatibility and the evolution of the SIS
- . definition of Extensibility and creation of a new term - Expandability
- . the notion of "conforming KAPSE".

Following the session, members were asked to rework section 6.4 of the R & C document the next day when they met in their working groups.

This was to take precedence and to be considered more important than completing KIWs.

26 JANUARY 1983

1. WORKING GROUP MEETING

The KIT met at TRW DSG San Diego facility. Working group meetings were held.

2. BREAK FOR LUNCH

3. RECONVENE

The KIT reconvened at TRW LSI facility for the afternoon session.

4. WORKING GROUP REPORT

The progress of the four working groups was presented by each of the four working group chairpersons. The highlights of the reports were as follows:

Working Group 1

Group 1 presented their "Bootstrap Mapse" model for moving tools from one APSE to another APSE. Their model assumed no tools on the target APSE.

Working Group 2

Group 2 proposed their model to port a project in which it moves a project's complete tool set. The group is attempting to get a good understanding of the SIS and its relationship to the KAPSE and the standard MAPSE tools.

Working Group 3

Group 3 worked on the following;

- . defined host (and not target) run-time requirements and criteria
- . defined requirements and criteria at the interface level for static and dynamic binding
- . changed "performance measurement" to "operational measurement"
- . determined interface considerations for operational measurement.

Working Group 4

Group 4 worked on separating SIS requirements from ways to implement a KAPSE. Other issues covered;

- . Extensibility vs Expandability
- . Security - should a DoD Instruction be cited as a requirement
- . interface considerations to security

Following the presentations by the chairpersons, a schedule for submitting updates to the Requirements and Criteria document was discussed whereby suggestions would be submitted and the document updated in time for the KITIA meeting. The reorganization of the updated document was also discussed.

5. STRATEGY

A strategy for the KIT was discussed. A plan to make the Requirements and Criteria document and the SIS Specification draft near-term, high priority usable products was proposed.

6. UPDATE TO STONEMAN

A document which updates the STONEMAN concept was presented. In the interest of time, all comments were withheld for the next day when the group would review it again.

27 JANUARY 1983

1. OPENING REMARKS

Tricia Oberndorf, the KIT chairperson, brought the meeting to order. She reviewed the agenda for the day and listed the day's four major topics of discussion; the KIT Strategy Paper, the SIS Draft, STONEMAN II and the I&T Requirements and Criteria Document.

2. KIT STRATEGY PAPER

Comments on the KIT Strategy paper were solicited. Responses are given below:

- . considerations as to what the paper represented, a policy or a strategy
- . the paper's relationship to the I&T Plan
- . the benefits of developing one and only one standard
- . the standardization process involvement (what needs to be done to make the document a government standard and to get it accepted by the AJPO and the three services)

- . the vehicle the AJPO must use to make the paper a policy and the mechanics required to publish it with respect to the environment
- . the available standards which could be used as a guide in formatting the paper for the standardization process
- . management of the standard
- . incorporation of a justification section in the paper to show the many things that were considered in producing the paper
- . enumeration of the parts of the APSE that are potential problems due to lack of experience
- . newly identified working groups to include Configuration Management, Policy and Validation
- . considerations for briefing senior service officials of the three services and a press release of the KIT activities for publication in software related magazines and journals
- . a refinement of the bulleted items in section 4; considered to comprise the heart of the document
- . refinement considerations for submission to the STI

Tricia will update the strategy paper, incorporate the appropriate suggestions and will have it finished for the upcoming KITIA meeting. At that time it will be available on the ARPANET through KIT-INFORMATION.

3. KIT/KITIA COMMUNICATION

A KITIA member Herb Willman (Raytheon) will attend future KIT SISWG meetings and will keep the KITIA informed as to the KIT's activities.

Jack Kramer (AJPO) assured the KIT that both the KIT and the KITIA are pursuing the same objectives as defined by phase 1 of the KITIA's 3 phase plan. Phase 1 plans call for a prototype standard to include a validation suite.

4. SIS DRAFT

The SIS draft was reviewed. The introduction and each of the five interface areas were addressed. Conversation concerned the document's organization and corrections that still needed to be made to the document. The document itself had "reader notice" notation where discrepancies and inconsistencies appeared. The highlights by interface area follow:

- . The Introduction will be expanded.
- . The Input/Output area was taken directly out of the Ada LRM. Exceptions and definitions for each package will be added.

- . A data base management model based on the UNIX structure was presented. Rationale for its structure was given. Follow-up discussion centered on access control considerations in particular security (need to know) to data bases.
- . A process management model developed and built to be analogous to the data base model was presented.
- . Each package in the Utilities section was discussed.
- . The contents of the MAPSE Tool Interfaces package are dependent on STONEMAN II progress.

KIT members were asked to review and comment on the document. A new method for collecting comments/suggestions for the SIS Draft (and any other KIT publication) is being considered whereby a separate ARPANET directory may be used to collect the information. KIT members will be kept informed as to its progress.

5. GENERAL

Following the SIS draft update the following topics were discussed:

- . distribution of the October minutes
- . SIS drafters to get together to decide when the next SIS meeting will be held
- . resolution of the two SIS acronyms
- . only one logo between the two groups (KIT & KITIA).

6. STONEMAN II

Tricia began by stating the purpose of the session; to define a set of standard interfaces to which APSEs can be written and therefore be made to support I&T.

The proposed revision to STONEMAN, called STONEMAN II, was presented. STONEMAN II attempts to redefine the MAPSE in two different areas; a transfer of functionality from the KAPSE data base to the MAPSE level and the elimination of the APSE data base.

Follow-up discussion concerned STONEMAN and its terminology, in particular, its definitions of APSE, MAPSE and KAPSE. Highlights of the discussion follow:

- . The current approach towards defining a MAPSE is a radical departure from the old STONEMAN which specified a minimum set of tools. The approach now is to define the common, standardized interfaces to tools.
- . The imprecise use of the notion of data base in STONEMAN.
- . The imprecise use by STONEMAN in the use of the word MAPSE with regard to the tool set. On some occasions STONEMAN refers to a

MAPSE as an Ada programming environment, on others, the MAPSE is used synonymously with the tool set.

- . The role of the program library and its implications to the MAPSE/KAPSE nomenclature.
- . The relationship of the SIS (Standard Interface Set) to the STONEMAN ideas of APSE, MAPSE and KAPSE.
- . For the purposes of discussion regarding the SIS, a motion was proposed to eliminate the words APSE, MAPSE and KAPSE and replace them with standard computer science terms.
- . A need was expressed to clarify STONEMAN document generalizations to prevent any mis-representation in future development efforts. A good example of what can happen by mis-representation is evidenced by the current dual APSE approach.

A new look at the APSE was proposed in which one looks into the implementation of the standard interface set through the tools or 'windows'. The approach shows how only the tools view the operating system that supports the APSE.

7. KIT PHOTO

A group picture of the KIT was taken.

8. BREAK FOR LUNCH

9. RECONVENE

10. MINUTES

The minutes of the October meeting were reviewed and approved as corrected.

11. STONEMAN II (Cont.)

Due to timing considerations, Tricia proposed that the STONEMAN II discussions be terminated.

It was stated that more work needs to be done on the document. Terms used in the document were not acceptable and new ones with more acceptable definitions needed to be introduced.

Again, a mail box type directory for receiving comments and suggestions concerning the STONEMAN II document will be set up. Any comments received by Tricia that have not been reviewed by the general KIT session will be distributed by her, to the KIT.

12. REQUIREMENTS AND CRITERIA DOCUMENT

Commencing with section 6.1.5 each section of the Requirements and Criteria document was reviewed and reworked (where appropriate). A synopsis of the comments follow:

- . Bulleted items under each subsection were examined to determine whether they were a requirement or criteria.
- . A reformat of the document was suggested where all requirements (and similarly all criteria) are grouped together under each subsection.
- . The word KAPSE was changed to 'SIS implementation'.
- . The definition and relation of the Ada environment to such items as default parameters, versatility/flexibility, exceptions and standard defined names were discussed.
- . The incorporation of new sections including Security and Evolution was discussed.
- . An examination of existing requirement documents such as STEELMAN was suggested as a way of providing additional input to the style and content of the R&C document.

Recommendations for updates to the R&C document are to be submitted via the ARPA Network to Tricia and/or Hal Hart.

13. SIS DEFINITION

Tricia opened the discussion by stating its objective; to decide where the SIS line should be drawn on the STONEMAN diagram. A discussion then started. A basic set of very elementary Ada tools were identified. Known collectively as the Required Ada Host Development System (RAHDS), this set of tools is considered the minimum set required to make it possible for other tools to be transferred.

Efforts continued in the expansion of thinking and definition of Interoperability and Transportability.

14. LOGOS

A new logo will be 'drawn up' by Tricia. It will be fashioned after a blank STONEMAN diagram.

15. ADJOURNMENT

Tricia Oberndorf, KIT chairperson, adjourned the KIT meeting. The next KIT meeting is scheduled for April 19 - 21 in Warminster, PA.

APPENDIX A

ATTENDEES

BALDWIN, Rich	U.S. Army CECOM
BASSMAN, Mitch	CSC
CASTOR, Jinny	AFWAL/AAAF-2
FERGUSON, Jay	NSA
FOIDL, Jack	TRW
FOREMAN, John	TI
FRENCH, Stewart	TI
FROMHOLD, Barbara	U.S. Army CECOM
HARRISON, Tim	TI
HART, Hal	TRW
HOUSE, Ron	NUSC
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	AJPO
KRUTAR, Rudy	NRL
LINDLEY, Larry	NAC
LOPER, Warren	NOSC
MAGLIERI, Lucas	National Defense Hqds.
MALONEY, Jim	Intermetrics
MILLER, Jo	NWC
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
NELSON, Eldred	TRW
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA

PURRIER, Lee	FCDSSA OOTB
ROBERTSON, George	FCDSSA OOTB
SCHAAR, Brian	AJPO
STEIN, Mo	NSWC/DL
TAFT, Tucker	Intermetrics
TAYLOR, Guy	FCDSSA
THALL, Rich	SofTech
WALTRIP, Chuck	Johns Hopkins Univ.

APPENDIX B

BIBLIOGRAPHY OF HANDOUTS

1. Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for Standard Interface Specification, 29 OCT 82, Hal Hart (TRW).
2. Agenda.
3. KIT minutes of October 1982 meeting.
4. KIT Strategy Paper, Tricia Oberndorf (NOSC).
5. Minimal Host for the KAPSE, 27 DEC 82, William Wilder (SofTech).
6. Proposed Standard Interface Set for an Ada Programming Support Environment, 26 JAN 83, Primary: Tucker Taft (Intermetrics).
7. Revised I&T Schedule.
8. STONEMAN II, Requirements for Ada Programming Support Environment, JAN 83, Donn Milton (Verdix).

MINUTES OF
KAPSE INTERFACE TEAM
INDUSTRY & ACADEMIA
21-23 FEBRUARY 1983
SAN DIEGO, CALIFORNIA

ATTENDEES: SEE APPENDIX A
MEETING HANDOUTS: SEE APPENDIX B

21 February 1983

1. WORKING GROUP MEETINGS

Edgar Sibley, KITIA Chairperson, brought the meeting to order. The KITIA reorganized into their respective Working Groups for discussions. This meeting marks the first anniversary of the KITIA.

22 February 1983

2. GENERAL MEETING

Edgar Sibley convened the general KITIA meeting. Lcdr. Brian Schaar (AJPO) announced that Ada is now an approved ANSI standard language (MIL-STD-1815A) as of 17 February 1983. The minutes of the October meeting were approved as corrected.

3. GENERAL DISCUSSIONS

Edgar Sibley opened the floor to discussion of the role of the KITIA. A summary of the major points follows.

- more technical interaction with the KIT is required including the insertion of KITIA points in KIT products and the distribution of KITIA papers to the KIT for comments and information.
- The KITIA should have a creative as well as a review role. Dit Morse sees the KITIA role as fourfold: 1) providing guidance to AJPO; 2) definition of activities to accomplish goals; 3) product review; and 4) performance of requested tasks.
- Working groups see their inputs in the context of their Working Group charters.
- The KITIA should be responsive to the KIT Standard Interface Set process.

Additional topics were discussed including:

- A change in the location of the July meeting from Cherry Hill to San Diego.
- The need for improvement in the documentation control process including meeting minutes and KITIA papers. Submission of papers via the Group chairs was recommended.

- A brief summary of the Murnau conference was presented that indicated KAPSE problems that were to be addressed in the STONEMAN II document.
- A directory of products to be included in the Public Report should be available on the ARPANET. The Group chairs are to meet with the KITIA chair and formulate a recommendation for the KITIA.
- The KITIA vice-chair should no longer be an elected position but should be appointed by the KITIA chair to assist in the formulation of agendas, document coordination and control, etc. This can also be a rotating position within the KITIA.
- There are problems regarding NET access. The group chairs are to coordinate the problems and report these to the AJPO.
- The KITIA would like the KIT to accept responsibility for the meeting arrangements for joint KIT/KITIA meetings.
- A COMMENTS directory is planned for the ARPANET similar to the KIT-INFORMATION. This is to present the comments received for the various KIT/KITIA documents. Additional data will be sent on the NET as progress continues.
- Anthony Gargaro presented a status report on the CONFIGURE tool. A configuration management workshop may be held in Falls Church in the near future. Additional data will be provided via the NET.

4. BREAK FOR LUNCH.

5. KITIA ELECTION

Edgar Sibley was the sole nomination for KITIA chairperson and elected unanimously.

6. KITIA PROPOSAL

Herm Fischer presented a three phase plan for execution of the KITIA proposal suggested at the last KITIA meeting. The first phase would address SIS requirements specification starting with the KIT Requirements and Criteria document and progressing to a B-5 like specification. This phase would also address the SIS interface specification starting with the KIT draft SIS and progressing to a C-5 like specification. The second phase would extend this work to a stable platform. The third phase would define a long term implementation task. A breakdown of the required organization and funding estimates to accomplish this task were also presented. A discussion of the merits and inherent problems in implementing this work was presented by Brian Schaar (AJPO). The KITIA was asked to provide an alternative method to implement the goals of their proposal. A special working group was formed under Dave McGonagle to examine this problem and prepare a plan.

7. WORKING GROUP MEETINGS

The KITIA reorganized into Working Groups to continue discussion and prepare future plans.

8. KITIA ADJOURNED FOR THE DAY.

23 February 1983

9. WORKING GROUP REPORTS

Working Group 1

- WG1 intends to prepare a Statement of Work for Phase 1 with emphasis on user services. This group intends to support the review/critique/normalization process, develop type A level requirements and develop type B level package specifications.
- In view of this effort the planned Command Language Workshop will be postponed. Additional meetings will be held 4/13, 4/14 in Blacksburg, 5/18-20 in Valley Forge, 6/15-17 in Cherry Hill, and 6/11-14 in San Diego.
- Review of the AIM and CONFIGURE tools will continue.

Working Group 2

- WG2 will continue their support of document reviews.
- This group feels the draft SIS implies a hierarchical file system which may be good for the AIE and ALS but not necessarily for a global SIS.

Working Group 3

- This group is looking at APSE/MAPSE/KAPSE boundaries for the standardization of their definition.
- The draft SIS review will continue with comments on the NET by 4 March.

Working Group 4

- This group intends to examine APSE/MAPSE/KAPSE in the context of SIS questions.
- A paper will be prepared on Distributed APSEs.
- KIT/KITIA document reviews will continue.

It was recommended that all APSE/MAPSE/KAPSE discussions have KITIA wide distribution.

Special SIS Working Group

- Dave McGonagle presented a plan in response to the AJPO request of the previous day for KITIA support to the SIS generation effort. This plan included augmentation of the KIT SIS Drafters with the KITIA Special SIS Working Group through a series of joint meetings resulting in a draft SIS for public review in the Fall of 1983.
- Deficiencies in the present draft SIS were identified with emphasis on the missing rationale.

- The cooperation of the entire KITIA in this effort was solicited. The KITIA voted overwhelmingly (YEA-22, Nay-0) to support this planned activity.

10. MEETING ADJOURNED

APPENDIX A
ATTENDEES
KITIA Meeting
21-23 February 1983

KITIA Members:

ABRAMS, Bernard	Grumman Aerospace Corp.
CORNHILL, Dennis	Honeywell/SRC
COX, Fred	Georgia Institute of Technology
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Systsems
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	Computer Sciences Corp.
GRIESHEIMER, Eric	McDonnell Douglas Astronautics
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles & Space
LAHTINEN, Pekka	Oy Softplan AB Finland
LAMB, J. Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
LOVEMAN, Dave	Massachusetts Computer Associates Inc.
McGONAGLE, Dave	General Electric
MORSE, H. R.	Frey Federal Systems
PLEODEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
RUBY, Jim	Hughes Aircraft Co.
SAIB, Sabina	General Research Corp.
SIBLEY, Edgar	Alpha Omega Group, Inc.
WESTERMANN, Rob	TNO-IBBC The Netherlands

WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

KITIA ALTERNATES:

BEANE, John	Honeywell
HUMPHREY, Dianna	Control Data Corp.

OTHER ATTENDEES

FOIDL, Jack	TRW
MYERS, Gil	Naval Ocean Systems Center
SCHAAR, Brian	Ada Joint Program Office

APPENDIX B
Bibliography of Handouts

I. Point Papers

- a. KAPSE Support for Program/Terminal Interaction

II. Minutes

- a. KITIA Minutes 4-5 October 1982
- b. KIT/KITA Joint Minutes 5 October 1982

III. Documents

- a. Ada Package Specification for the Standard Interface Set
1 February 1983 (Draft)
- b. KIT Strateg Statement
25 January 1983 (Draft)
- c. STONEMAN II
January 1983 (Draft)
- d. Ada Programming Support Environment (APSE) Requirements for
Interoperability and Transportability and Design Criteria for
Standard Interface Sets
18 February 1983 (Working Paper)

IV. OTHER

- a. Revised I & T Schedule

KIT/KITIA MINUTES
MEETING OF 18-21 APRIL 1983
WARMINSTER, PENNSYLVANIA

ATTENDEES: SEE APPENDIX A
BIBLIOGRAPHY OF HANDOUTS: SEE APPENDIX B

18 APRIL 1983 - KITIA MEETING

1. OPENING REMARKS

Edgar Sibley, KITIA chairperson, brought the meeting to order.
Larry Johnston, Naval Air Development Center, was introduced as the local host.

2. GENERAL BUSINESS

KITIA members were asked to recommend the best time for the joint October KIT/KITIA meeting through their respective Working Group chairs.

A recommendation was made to have the KITIA rules available through the ARPANET for members access.

Distribution of lengthy text via ARPANET messages may be facilitated through identification of a users file name so that interested parties may copy the file for printout rather than generation of multi-page mail messages. A NET message of a few lines may then be transmitted using Group lists to identify the appropriate file.

3. WORKING GROUP REPORTS

Working Group 1 - continues review of KIT/KITIA documents.

Working Group 2 - continues review of KIT/KITIA documents.

Working Group 3 - submitted comments on the Standard Interface Set document; preparing a proposal to modify the KIT/KITIA strategy document.

Working Group 4 - in the process of electing a new Working Group chairman.

SIS Working Group (SISWG) - held two joint meetings with KIT drafters in Boston area. Held a KITIA SISWG meeting in Palo Alto. Working toward a liveable SIS that captures the STONEMAN goals. Work has effectively merged the KIT/KITIA SIS efforts into a single team effort. Continuing progress on data management, process control, and virtual terminal concepts. Definition of Rationale section progressing. Future meeting in Boston area will continue.

4. AJPO STATUS

Lcdr. Brian Schaar of the Ada Joint Program Office reported that the KAPSE development work on the Ada Integrated Environment has been halted by the Air Force. The work on the compiler and compiler-related tools is continuing. The problem was caused by lack of Government Furnished Equipment requiring the developer to incur unbudgeted computer lease expenses. Plans for the re-start

of the KAPSE work were not yet available. The Ada Language System is progressing according to its new development plan and schedule. A detailed schedule should be available by the next KIT/KITIA meeting.

5. GENERAL DISCUSSION

The latest version of the SIS does not appear different from the previous version. Why?

This version reflects a re-organization agreed to by the KIT drafters but a major revision is not planned until the KITIA SISWG is integrated into the team and their contributions can be effectively assimilated. The next version will contain these changes. All members were requested to provide "constructive criticism" versus "criticism" when reviewing the KIT/KITIA documents.

6. BREAK FOR WORKING GROUPS

The KITIA members reorganized into individual working groups.

7. RECONVENE FOR ADDITIONAL DISCUSSION

The KITIA was requested to review the Minutes of the previous meeting and report any problems to their Group chairs.

Suggestions were made for establishment of document review files on the ARPANET. These include a Version Description Document format to identify significant revisions since the previous version, responses to individual comments in the comment file (including status such as incorporated, deferred, or rejected), and the inclusion of change bars to reflect present changes in the various documents.

8. ADJOURN FOR THE DAY.

19 APRIL 1983 - JOINT KIT/KITIA MEETING

1. OPENING REMARKS

Gil Myers, acting KIT chairperson, brought the meeting to order.

2. GENERAL BUSINESS

Lcdr. Brian Schaar, Ada joint Program Office, announced that Jack Kramer will continue working on the Ada Program as an employee of the Institute for Defense Analysis. The AJPO has also initiated a new task directed at the Evaluation and Validation of APSEs. The initial work is being supported by the Air Force. The status of the Standard Interface Set document and the integration of the KITIA members into a joint SIS Working Group was described.

Edgar Sibley, KITIA chairperson, presented the highlights of the previous day's KITIA meeting including the KITIA initiative to become more actively involved in the SIS. He suggested the KIT and KITIA Working Group chairs establish closer working relations. Dr. Sibley also reminded all attendees of the sensitivity of the documents under development and cautioned against unauthorized distribution without prior approval of the AJPO.

The KITIA Working Group chairs reported the progress of their respective working groups. The SISWG will provide additional details later in the meeting.

Jack Foidl, TRW, described the document review methodology that is being established. This includes following the review comment format utilized in the AIE document review, serialization of all KIT and KITIA products, establishment of a COMMENTS review directory on the ARPANET, and the submission of materials for inclusion in the Public Report.

Mitch Bassman, CSC, presented the status of the CONFIGURE Tool. The design of the tool will be completed but will not be developed due to the non-availability of a validated compiler. Additional options are being examined.

John Foreman, TI, reported on the status of the AIM Tool and thanked those that participated in the reviews of the AIM documentation. The interaction with the SISWG regarding terminal interfaces was also reported.

Hal Hart, TRW, presented an preview of topics to be presented at NCC '83 in Anaheim, California in May.

Anthony Gargaro, CSC, presented highlights of his presentation to the Brussels conference on behalf of the AJPO.

3. EUROPEAN Ada/APSE STATUS

Erhard Pleodereder, IABG West Germany, presented an overview of their efforts which is basically sponsored by their Ministry of Defense. Work includes development of a front end (which is currently based on Ada-80), two back ends and a symbolic debug system based on DIANA. Their work will be

impacted by ANSI Ada.

Tim Lyons, Software Sciences Ltd., England, presented an overview of the United Kingdom efforts. The work is directed under Ada Group Ltd. which is a consortium of three companies developing MCHAPSE, a Minimal Chill and Ada Programming Support Environment for the U.K. Telecom with some Ministry of Defense funding. This group expects to have a compiler by the end of 1984 and an environment in the post 1984 time frame. The initial target is an ICL 2900 with the VAX/VMS as the second target. A description of the KAPSE and database is contained in the U.K. Study Reports. This effort is currently planned at approximately \$13M.

4. STANDARD INTERFACE SET WORKING GROUP

Dave McGonagle, General Electric, presented the status of the SISWG. The initial efforts have been to incorporate KITIA concepts and ideas into the initial KIT SIS document. The ground rules of this initial effort have been to formulate a SIS that is livable for the AIE and ALS, captures STONEMAN concepts, and is useable. Members of the SISWG were introduced and a summary of the progress to date identified. Future potential problem areas were presented including data management, process control, and extent of virtualization of the terminal interfaces.

5. BREAK FOR LUNCH

6. STANDARD INTERFACE SET WORKING GROUP (Cont'd)

Eli Lamb presented the results of definition of the SIS Rationale to be included in the SIS. The SIS was defined as a standard set of host independent interfaces providing access to system services. Rationale for the various SIS sections will be included as they are identified. A discussion of the relationship of the Requirements and Criteria document to the SIS resulted in clarification of the RC document as applicable to the future SIS and not the initial SIS. Also, the initial SIS should not constrain the composition of the RC document.

[NOTE: the following table is provided for the readers understanding of the context of the various terms that were used in these discussions.

Initial SIS	SIS 0	SIS 84	- the SIS applicable to the AIE/ALS environments
Final SIS	SIS 1	SIS 86	- the SIS to be promulgated as a Military Standard for DoD environments]

The data management model as included in the initial SIS was presented by Tucker Taft. A different perspective was presented by Erhard Pleodereder. A discussion of the various merits of each model ensued.

A Strategy for Advancement of the Standard Interface Set was recommended. Possible test cases for a meaningful SIS such as a Command Interpreter and a Virtual Terminal were also presented.

7. VIRTUAL TERMINALS

Perspectives on virtual terminal concepts were presented by Stewart French of Texas Instruments and Herm Fischer of Litton Data Systems.

8. BREAK FOR WORKING GROUP MEETINGS.

9. ADJOURN FOR DAY

20 APRIL 1983

10. JOINT WORKING GROUP REPORTS

Working Group 1

- continuing review of the Requirements and Criteria document to which the following observations apply:

Introduction

- need a definition of a "conforming SIS"
- missing focus on interoperability
- references to KIT activities should be placed in a cover letter
- what is the distinction between "serious tool writers" and "writers who are serious about I&T"

Section 3 (prior to 3.4.1)

- we need to clearly distinguish between guidelines, criteria and requirements. Guidelines shall not be called criteria. All criteria should be quantifiable. Guidelines may be included in a separate document

Section 3.4.1

- Rationale is needed by section and by bullet
- definitions are required for program, process and user (clarify 'user' of SIS or APSE)
- requirements should be testable, i.e., "Facilities...for a user to redirect program input and output" could be satisfied by two different SISs
- need a privilege request and access granting mechanism that is dynamic
- in statements such as "the calling task of the caller waits" the SIS process facilities must know about Ada tasks and there may be hidden implications
- there must be a clear distinction between the requirements on interfaces and the requirements on facilities supporting the interfaces

- the joint KIT/KITIA working group coordination raises problems associated with travel so the groups may organize into subsets to address specific topics

- continuing SOW defined in San Diego in February

Working Group 2

- the KITIA WG spent Monday in a review of STONEMAN II and the RC document and had the same problems as WG1 with "serious" and the "interoperability" issue

- the Joint WG performed some basic review of the I/O section as well as the database management and control areas
- this WG is having some trouble with definitions such as the SIS database. To what level do you address and to what depth is required in the SIS?

Working Group 3

- D. Wrege is working on a SIS related paper for the Public Report
- this WG will provide some recommendations on the structure of the SIS
- will try to concentrate on areas that are separated by needs such as SIS 0, the RC document, and definition of a Standard Ada Library (such as math pacs)
- there is a need for more criteria since the SIS is at the boundary of tool sets and the environment; the information transfer is in and out of a program's address space, therefore, identification of facilities that must or must not be standardized will be required
- future work of this group includes:
 - Drake/Saib to formulate a matrix of tools/interfaces
 - Fellows to define a process model for the SIS
 - Lamb/Willman to continue SIS drafter work
 - Johnson to work on the RC document

Working Group 4

- organizing into a number of sub-groups
- defining future plans of action to support
 - interoperability issues
 - user interfaces
 - METHODMAN liaison
 - inter-tool data dependencies (and other DIANA-like structures)
 - conventions and guidelines
 - standard run-time services (incl. host vs. target issued)
 - RC document
 - STONEMAN II
 - Evaluation & Validation liaison
 - SIS
 - Policy
 - Standard Glossary

11. KITIA MINUTES

The KITIA Minutes were approved as corrected. Corrections included AdaTEC typo correction and addition of Texas Instruments AIM presentation.

12. OCTOBER MEETING DATES

The October meeting is tentatively scheduled for 17-19 October 1983 in

Dallas.

13. AJPO CLOSING COMMENTS

Lcdr. Brian Schaar expressed pleasure with the joint teams' progress. He reflected that the pressure for the SIS was increasing and there remains a great deal of work to be accomplished. The industry perspective provided by the KITIA participants is especially valuable in this hard endeavor.

14. JOINT KIT/KITIA MEETING ADJOURNED

21 APRIL 1983 - KIT MEETING

1. DOD SESSION

2. TREASURER'S REPORT

- an accounting of the registration monies was presented

3. KIT MEETING MINUTES

- the KIT meeting minutes from January were to be revised and presented at the next meeting for approval.

4. KIT MEETING ADJOURNED

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
April 1983

KIT Members:

BASSMAN, Mitch	Computer Sciences Corporation
CASTOR, Jinny	AFAWL/AAAF-2
DUDASH, Ed	NSWC/DL
FERGUSON, Jay	NSA
FOIDL, Jack	TRW
FOREMAN, John	Texas Instruments
FROMHOLD, Barbara	U.S. Army CECOM
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
HOUSE, Ron	NUSC
JOHNSON, Doug	SoftWrights
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	Institute for Defense Analysis
KRUTAR, Rudy	NRL
LINDLEY, Larry	NAC
LOPER, Warren	NOSC
MAGLIERI, Lucas	National Defense Hdqs., Canada
MILLER, Jo	NWC
MOLONEY, Jim	Intermetrics
MYERS, G11	NOSC
MYERS, Phil	NAVELEX
PEELE, Shirley	FCDSSA, Dam Neck
ROBERTSON, George	FCDSSA, San Diego
SCHAAR, Brian	AJPO

STEIN, Mo	NSWC/DL
STOPYRA, Norma	NAV MAT-08Y
TAFT, Tucker	Intermetrics
TAYLOR, Guy	FCDSSA, Dam Neck
THALL, Rich	SofTech
WALTRIP, Chuck	John Hopkins Univ.

KITIA Members:

ABRAMS, Bernie	Grumman Aerospace
BAKER, Nick	McDonnell Douglas Astronautics
CORNHILL, Dennis	Honeywell/SRC
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Sytsems
FREEDMAN, Roy	Hazeltine Corp.
GAJNAK, George	Alpha Omega Group
GALLAHER, Larry	Georgia Institute of Technology
GARGARO, Anthony	Computer Sciences Corp.
GLASEMAN, Steve	Teledyne
JOHNSON, Ron	Boeing Aerospace
KERNER, Judy	Norden Systems
LAMB, J. Eli	Bell Labs
LYNDQUIST, Tim	Virginia Institute of Technology
LYONS, Tim	Software Sciences Ltd., United Kingdom
McGONAGLE, Dave	General Electric
MORSE, H. R.	Frey Federal Systems
PLEODEREDER, Erhard	IABG, West Germany
REEDY, Ann	PRC
RUBY, Jim	Hughes Aircraft Co.
SIBLEY, Edgar	Alpha Omega Group, Inc.
WESTERMANN, Rob	TNO-IBBC, The Netherlands
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
Yelowitz, Larry	Ford Aerospace

APPENDIX B
Bibliography of Handouts

I. Point Papers

- a. Rationale for a Standard Interface Specification
(SISWG Working Draft)

II. Minutes

- a. KITIA Minutes 21-23 February 1983
- b. KIT Minutes 25-27 January 1983

III. Documents

- a. Ada Package Specification for the Standard Interface Set
15 April 1983 (Draft)
- b. KIT Strategy Statement
25 January 1983 (Draft)
- d. Ada Programming Support Environment (APSE) Requirements for
Interoperability and Transportability and Design Criteria for
Standard Interface Sets
15 April 1983 (Working Paper)

Minutes of the
SIS Drafters Meeting
8-9 March 1983
Waltham, Massachusetts

ATTENDEES: Lcdr. B. Schaar - Ada Joint program Office
Gil Myers - Naval Ocean Systems Center
KIT SIS Drafters

J. Foidl

J. Kramer

T. Taft

R. Thall

W. Wilder

KITIA SIS Drafters

E. Lamb

D. Mc Gonagle

H. Willman

Invited guest: S. French

HANDOUTS: Schedule of KITIA SIS Activities
Comments for SIS Introduction from P. Oberndorf
KITIA SIS document format outline
Texas Instruments "Virtual Terminal" presentation hardcopy

=====

TUESDAY 8 March 1983

1. Introductory comments by B. Schaar regarding KITIA efforts in the development of the SIS. Reviewed background of KITIA Proposal and the formulation of the KITIA SIS Working Group.

2. General discussion were conducted regarding:

- Process model and history model
- relational versus hierarchical data structures (relational type has not been excluded as yet)
- formulated basic SIS inclusion test utilized to date:
 1. Can you obtain a reasonable implementation on the AIE?
 2. Can you obtain a reasonable implementation on the ALS?
 3. Can you obtain a reasonable implementation on a modern operating system (as reflected by UNIX, VMS, TOPS-20 time-share systems)?
 4. Can you obtain a reasonable implementation on a bare machine?
- use of special terms/formats as "reserved" for top level descriptors such as TOOLS.XXX
- concept of importing/exporting not fully explored
- discussion of RENAME/LINK as required/handy respectively
- implementation expense of inheritable attributes in tree structure
- although database and process control are similar there may be differences in their control features
- pointer control in NEXT/INDEX functions

3. Presentation by S. French of Texas Instruments regarding their Virtual Terminals concept. Counterpoint implementation via an extension of TEXT IO suggested by T. Taft. Discussion revolves around the level of the interfaces

that are required in the SIS.

WEDNESDAY, 9 March 1983

4. KITIA SIS Drafters joined the discussions. A review of previous SIS work and discussion of rationales followed.

- concern with the implication of a file system as the SIS basis
- recommendation the KITIA SIS Drafters designate one ALS and one AIE expert as future point of contact.
- missing interface points for debugging support, performance measurement, help, and security
- although a strict hierarchy may be required at this time, this may not be a firm requirement in the future
- cursors may have to be passed in a package

5. Discussion of the Process model.

- spawn job cannot be fully implemented at this time
- old Text Manager function from Ada LRM be reviewed for inclusion in next SIS draft
- SIS Utilities may fall under Inter-Tool Communication
- list processing requires further analysis
- set ordering needs examination; ASCII lexicographical for now
- interrupts provided but not currently available in ALS

6. Discussion of future activities.

- All future SIS meetings to be held in Boston area
- Next SIS Drafters meeting 6-7 April 1983.

7. Meeting adjourned.

SECTION 3

KIT/KITIA DOCUMENTATION

Ada Programming Support Environment
(APSE)
Interoperability and Transportability (I&T)
Management Plan

15 December 1982

for

Ada JOINT PROGRAM OFFICE
The Pentagon
Washington, D.C. 20301

prepared by

NAVAL OCEAN SYSTEMS CENTER
271 Catalina Boulevard
San Diego, California 92152

1.0 INTRODUCTION

The Ada Programming Support Environment (APSE) Interoperability and Transportability (I&T) Plan is presented in this document. The I&T activities necessary to achieve sharing of tools and data bases between APSEs are described. Schedules and milestones for these activities are presented as well as a Work Breakdown Structure (WBS) for accomplishing them.

These I&T activities are conducted by the Kernel APSE Interface Team (KIT).

The major responsibilities are:

- a. APSE I&T Management
- b. APSE I&T Analysis
- c. APSE I&T Standards Development
- d. APSE I&T Tools Development
- e. APSE I&T Coordination with Implementation Efforts

1.1 BACKGROUND

In 1975 the High Order Language Working Group (HOLWG) was formed under the auspices of the U.S. Department of Defense (DoD) with the goal of establishing a single high order language for new DoD Embedded Computer Systems (ECS). The technical requirements for the common language were finalized in the Steelman report [1] of June 1978. International competition was used to select the new common language design. In 1979 the DoD selected the design developed by Jean Ichbiah and his colleagues at CII-Honeywell Bull. The language was named Ada in honor of Augusta Ada Byron (1816-1851), the daughter of Lord Byron and the first computer programmer.

It was realized early in the development process that acceptance of a common language and the benefits derived from a common language could be increased substantially by the development of an integrated system of software development and maintenance tools. The requirements for such an Ada programming environment were stated in the STONEMAN document [2]. The STONEMAN paints a broad picture of the needs and

[1] Requirements For High Order Computer Programming Languages: STEELMAN, DoD, June 1978

[2] Requirements for Ada Programming Support Environments, STONEMAN, DoD, February 1980

identifies the relationships of the parts of an integrated APSE. STONEMAN identifies the APSE as support for "the development and maintenance of Ada application software throughout its life cycle". The APSE is to provide a well-coordinated set of tools with uniform interfaces to support a programming project throughout its life cycle. The Initial Operational Capabilities (IOCs) are called Minimal Ada Programming Support Environments (MAPSEs).

The Army and Air Force have begun separate developments of APSEs. The Army APSE has been designated the ALS (Ada Language System) and that of the Air Force, the AIE (Ada Integrated Environment). The Navy APSE will make maximum use of those Army and Air Force products that meet Navy requirements and will require the development of only those additional components required for Navy applications.

The Ada Joint Program Office (AJPO) was formed in December 1980. The AJPO coordinates all Ada efforts within DoD to ensure their compatibility with the requirements of other Services and DoD agencies, to avoid duplicative efforts, and to maximize sharing of resources. The AJPO is the principal DoD agent for development, support and distribution of Ada tools and Ada common libraries.

1.2 DEFINITIONS

INTEROPERABILITY: Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.

TRANSPORTABILITY: Transportability of an APSE tool is the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonyms.

1.3 OBJECTIVES

The objectives of the APSE I&T effort are:

- a. To develop requirements for APSE I&T.

STONEMAN paints a broad picture of the needs and relationships of the parts of an integrated APSE. Although STONEMAN is being used as

the primary requirements document for APSE development efforts, it does not provide sufficient detail to assure I&T between APSEs. APSEs built to accomodate I&T requirements will insure cost savings in the development of tools. The cost of re-programming tools for different APSEs will be significantly reduced.

b. To develop guidelines, conventions and standards to be used to achieve I&T of APSEs.

Guidelines, conventions, and standards describe the means by which the requirements can be satisfied. It would be premature to develop steadfast standards during the early part of this APSE I&T effort. There is little precedent for I&T between programming support environments of this anticipated magnitude and thus little guidance for the development of these guidelines, conventions, and standards. The guidelines, conventions and standards that are developed during this APSE I&T effort will evolve over a four year period from 1982 to 1985. These guidelines, conventions, and standards will be presented in public forums to insure that they are sound and realistic.

c. To develop APSE I&T tools to be integrated into both the AIE and ALS.

This APSE I&T effort provides for the development of three or more tools to be integrated into both the AIE and the ALS. These tool development efforts will help identify interfaces and surface interface problems associated with I&T between different APSEs. They should also show how closely the guidelines, conventions and standards developed by this APSE I&T effort reflect the reality of the AIE and ALS efforts. But the tools developed by this APSE I&T effort will not be limited to this test function. They will also be well documented tools which will become useful additions to any APSE.

d. To monitor the AIE and ALS development efforts with respect to APSE I&T.

This APSE I&T effort provides for the monitoring of the AIE and ALS development efforts. The monitoring will result in recommendations for resolution of differences between the AIE or the ALS and the evolving APSE I&T conventions and standards. Interface areas which would inhibit I&T between the AIE and ALS will also be identified.

AIE and ALS documents will be reviewed and analyzed, and recommendations will be made. When questions arise that need resolution and/or clarification with regard to the ALS and AIE

development efforts the KIT (see Section 2.3) will rely on the assistance of Army and Air Force members who are involved in these efforts.

e. To provide initiative and give a focal point with respect to APSE I&T.

A focal point is needed for APSE developers and users with regard to information about I&T. APSE I&T questions arise frequently within professional societies and user groups. A forum is needed in which APSE I&T questions can be addressed and discussed and in which APSE I&T information can be disseminated throughout the Ada community.

The KIT and KITIA (see Sections 2.3 and 2.4) will provide focal points for the Ada community. Public reports on the results of this APSE I&T effort will be published every six months. This is in keeping with the AJPO philosophy of public exposure of all aspects of the Ada program. The KIT and KITIA will also participate in other programs connected with APSE I&T, including international development efforts, whenever possible.

f. To develop and implement procedures to determine compliance of APSE developments with APSE I&T requirements, guidelines, conventions and standards.

Procedures must be established by which the recommendations that are developed by this APSE I&T effort will be reviewed and implemented by the AJPO. The procedures that are to be followed should apply not only to the AIE and ALS development efforts, but also to other APSE development efforts. Work on the determination of compliance procedures will be pursued in cooperation with the AJPO's Evaluation and Validation program.

1.4 DOCUMENT ORGANIZATION

Section 1 of this document discusses the purpose and scope of the I&T Plan, the objectives of the I&T effort, and the basic concepts, definitions, and objectives.

Section 2 discusses the sponsorship, the participating organizations, the organizational inter-relationships and responsibilities, and the potential forums for public involvement.

The specific tasks to be accomplished in pursuit of I&T are covered in Section 3. These functions are presented in a work breakdown structure for the project and a schedule of milestones and deliverables.

Special needs in achieving I&T are discussed in Section 4.

Appendix A contains a glossary of terms and acronyms applicable to the I&T effort and Appendix B contains a bibliography of AIE documents. Appendix C contains a bibliography of ALS documents and Appendix D contains other APSE related documentation. Appendix E describes the elements of the I&T Work Breakdown Structure.

2.0 ORGANIZATION

Figure 1 shows the participants in the APSE I&T effort. The following sections provide a brief description of these organizations and their relationships.

2.1 Ada Joint Program Office

The KIT is an agent of the Ada Joint Program Office (AJPO). The KIT supports the AJPO by performing the activities outlined in this plan and by providing recommendations and information to the AJPO. The AJPO makes final decisions in the areas of requirements, policy, procedures and funding.

2.2 ARMY, AIR FORCE AND NAVY

Currently the Army and Air Force have begun separate developments of APSEs. In the development of its APSE, the Navy plans to make maximum use of Army/Air Force products that meet Navy requirements. The KIT will review of all these APSE developments and identify critical aspects of the designs where conventions or standard interfaces and specifications are needed to insure compatibility. It will be the role of the KIT to interact with these services and their respective APSE contractors for information-exchange and consultation. The contractor for the Army's ALS is SofTech Inc.; the Air Force contractor for the AIE is Intermetrics Inc.. The Navy contractor has not been selected yet. Representatives of both the Air Force and Army APSE development efforts are members of the KIT, and many members of the Navy's Design Review Group (DRG) serve on the KIT as well.

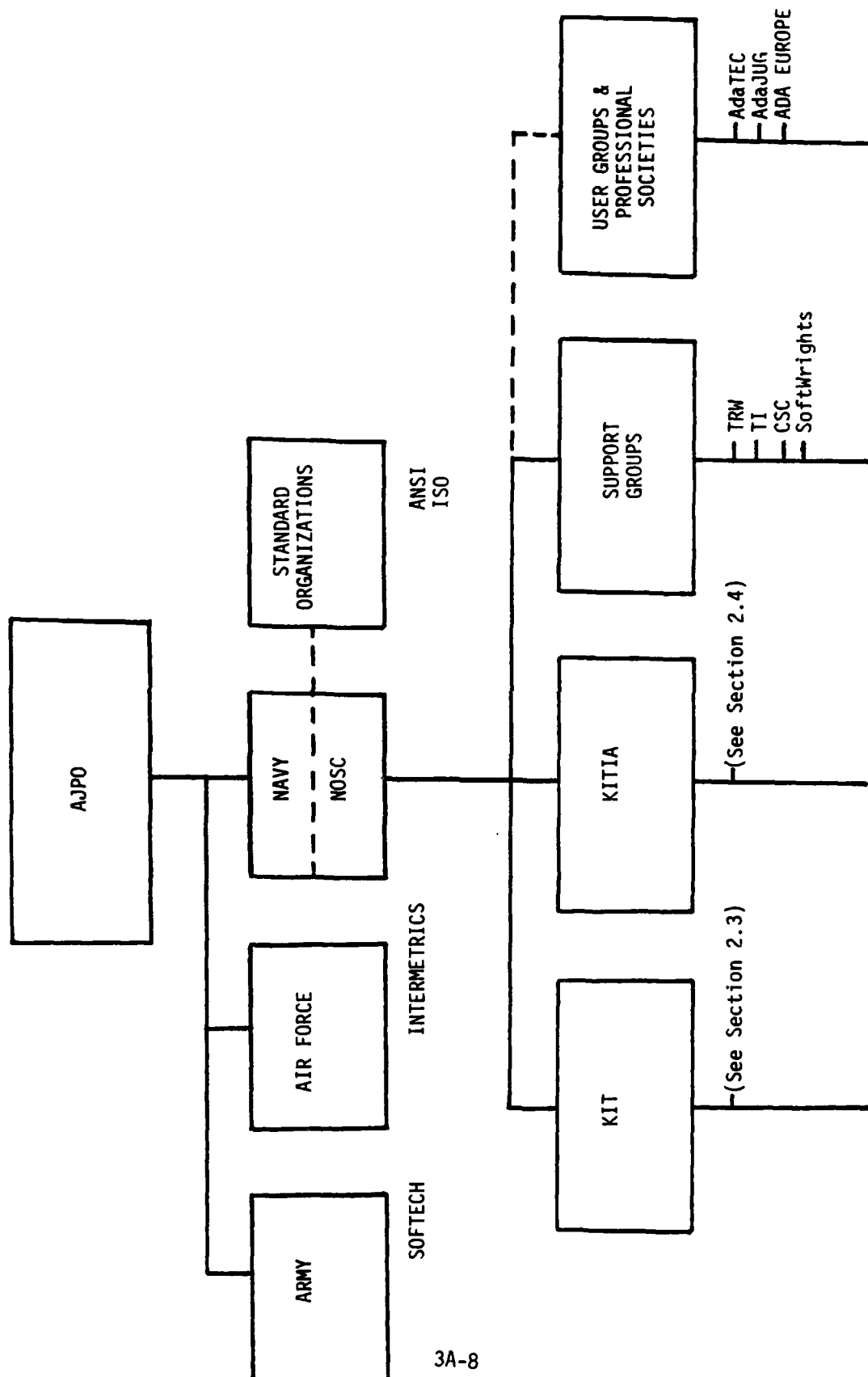


Figure 1 APSE IT Participants

2.3 KAPSE INTERFACE TEAM (KIT)

The objectives of the KIT are the objectives of the APSE I&T effort (see Section 1.3). The Navy is responsible for chairing the KIT. The membership is composed of the following representatives:

- Naval Ocean Systems Center (NOSC)
- Naval Sea Systems Command (NAVSEA/PMS-408)
- Naval Electronics Systems Command (NAVELEX)
- Naval Underwater Systems Center (NUSC)
- Naval Surface Weapons Center (NSWC)
- Naval Avionics Center (NAC)
- Naval Air Development Center (NADC)
- Naval Research Laboratory (NRL)
- Fleet Combat Direction System Support Activity (FCDSSA) - Dam Neck
- Fleet Combat Direction System Support Activity (FCDSSA) - San Diego
- U.S. Air Force - Rome Air Development Center (RADC)
- U.S. Air Force - Air Force Wright Aeronautical Laboratories (AFWAL)
- U.S. Army - Communications and Electronics Command
- U.S. Air Force - Information Processing Standards for Computers (USAF-IPSC)
- Johns Hopkins University Applied Physics Laboratory (JHUAPL)
- National Security Agency
- Canadian National Defense Headquarters

NOSC is the Navy laboratory which provides the KIT chairman. All other members participate on a volunteer basis, aided as necessary by the AJPO with funding for such things as travel expenses. New members will be added to the KIT at the discretion of the AJPO.

Because of the potentially large membership of the KIT, a management steering committee called the KIT Executive Committee (KITEC) has been established. It consists of the AJPO sponsor (i.e., the AJPO Navy deputy), the KIT chairman, the primary support contractor (see Section 2.5), and selected other KIT members as determined by the sponsor and chairman. The KITEC is responsible for the planning and management of the APSE I&T effort, including maintenance of this plan and direction of activities in accordance with its tasks and schedules.

In addition, the KIT is divided into various working groups for the purpose of small group concentration on specific technical areas affecting I&T. The number, objectives, and membership of such working groups may change as KIT needs change.

2.4 KAPSE INTERFACE TEAM FROM INDUSTRY AND ACADEMIA

The KITIA was formed to compliment the KIT and to generally contribute a non-DoD perspective to the I&T effort. The KITIA supplements the activities of the KIT. It assures broad inputs from software experts and eventual users of APSE's. The KITIA interacts with the KIT as reviewers, as proposers of APSE I&T requirements, guidelines, conventions and standards, and as consultants concerning implementation implications. The team was selected from applicants representing industry and academia. The following are the members of the KITIA:

- Alpha-Omega Group
- Bell Laboratories
- Boeing Aerospace
- Computer Sciences Corporation
- Control Data Corporation
- Ford Aerospace
- Frey Federal Systems
- General Electric
- General Research
- Georgia Institute of Technology
- Grumman Aerospace
- Hazeltine
- Honeywell
- Hughes Aircraft
- IBM
- Litton
- Lockheed
- McDonnell Douglas
- Norden
- PRC
- Raytheon
- SDC
- Teledyne
- TNO (The Netherlands)
- UK Ada Consortium
- Virginia Polytechnic Institute

In addition, the following have been asked to be special associate members of the team:

- IABG (W. Germany)
- Massachusetts Computer Associates
- Oy Softplan Ab (Finland)
- University of California at Irvine

Membership on the team belongs to a company or university, and not to an individual representing his/her organization. All participation

is voluntary, and the members selected have agreed to provide 1/3 of a man-year plus other support such as travel expenses. The membership of the KITIA will not be expanded unless an organization withdraws or very special circumstances apply. The AJPO sponsor and KIT chairman are ex officio members of the KITIA.

The KITIA elects a chairman and a vice-chairman from amongst its participants every year. It, too, is organized into working groups who in turn select their own chairmen. The KITIA chairman and vice-chairman together with the working group chairmen form the KITIA management committee.

The KITIA is responsible to the AJPO through the KIT chairman. Although the KIT has ultimate responsibility for the development of all products required to meet the I&T objectives, the KITIA participates directly in the generation and review of such products. In addition, the KITIA generates its own contributing papers, products, initiatives, and recommendations to supplement and guide the basic KIT efforts. This requires close coordination, which is facilitated by ARPANET communication mechanisms, parallel working group structures, and joint team meetings.

2.5 SUPPORT CONTRACTORS

Currently there are four contractors that participate on the KIT. TRW is the primary support contractor, providing general support and technical initiatives. Texas Instruments and Computer Sciences Corporation are developing APSE tools in support of the I&T objectives (see Section 1.3c). One or more additional contractors will develop additional APSE I&T tools. Mr. Doug Johnson provides overall review and consultation for the AJPO.

Any of these contractors may also serve as a vice-chairman of a KIT working group.

2.6 USER GROUPS AND PROFESSIONAL SOCIETIES

It is anticipated that AdaTEC, the JOVIAL-Ada Users Group (JUG), and Ada Europe will provide valuable contributions to the APSE I&T effort. The KIT and KITIA have no formal relationship with these groups; however, the KITEC will use some or all of these groups as regular forums for the presentation of reports and technical results and will solicit feedback from their members.

2.7 STANDARDS ORGANIZATIONS

The American National Standards Institute (ANSI) and the International Standards Organization (ISO) are standards organizations which are already involved in establishing the Ada programming language as a broadly recognized, enforceable standard. It is possible that the results of this I&T effort will be submitted for such approval by these organizations as well, to effect the commonality of APSE's deemed necessary to achieve DoD's life-cycle objectives. The KIT initially will become familiar with the organizations' standardization procedures so that future standardization actions can be planned and accomplished with minimum difficulty. This will include the study of existing standards which may interact with or guide the development of, APSE I&T standards.

2.8 LAISON WITH IMPLEMENTATION EFFORTS

A number of implementation efforts have been undertaken by organizations outside of the DoD. Three of these (the U.K. Ada Consortium, the West German IABG and U.C. Irvine) are represented on the KITIA. Others include the European Economic Community, ROLM Corporation, Western Digital, and Telesoft, just to name a few. The KIT will keep such organizations informed of its activities and will consider all feedback received from them.

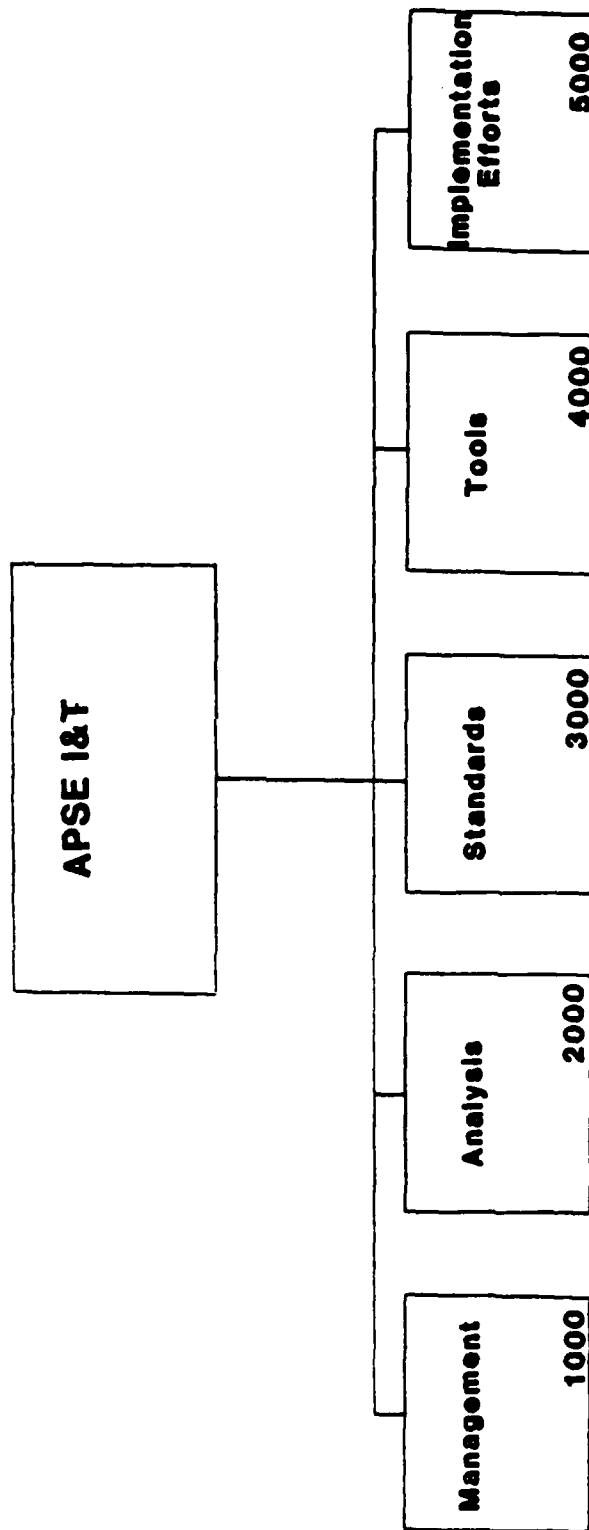


Figure 2 WBS Overview

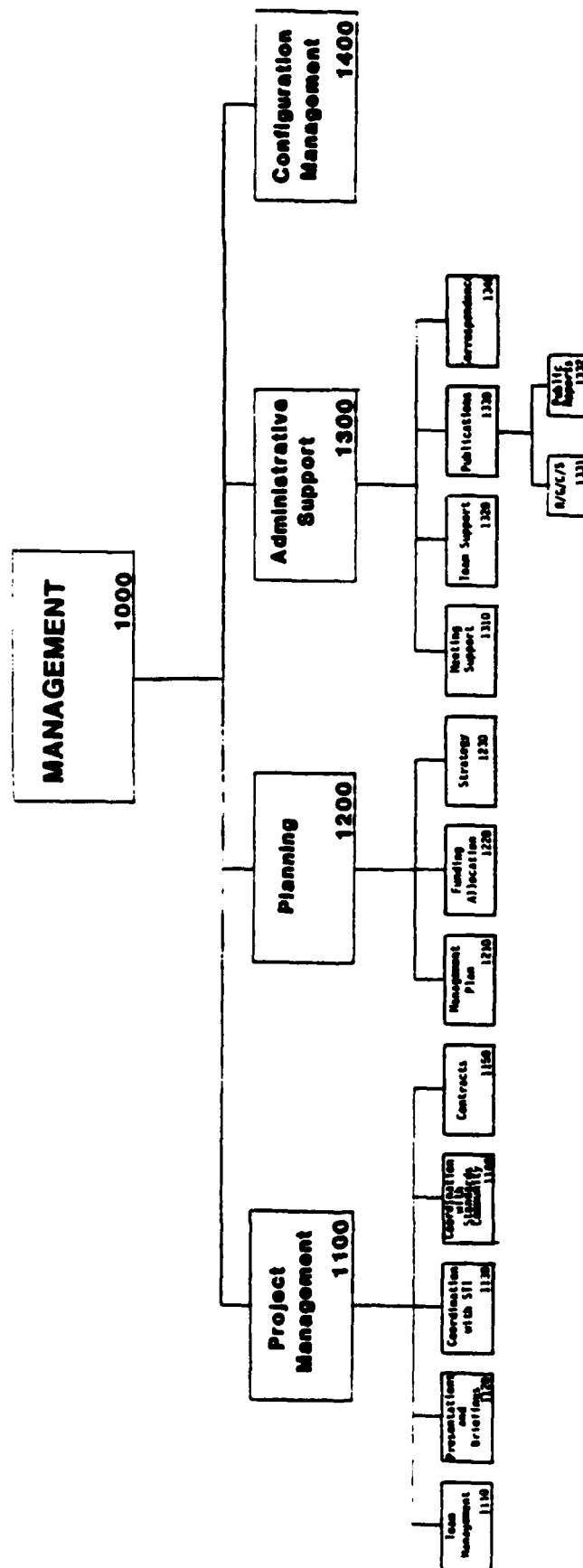


Figure 3 IBS Management Overview

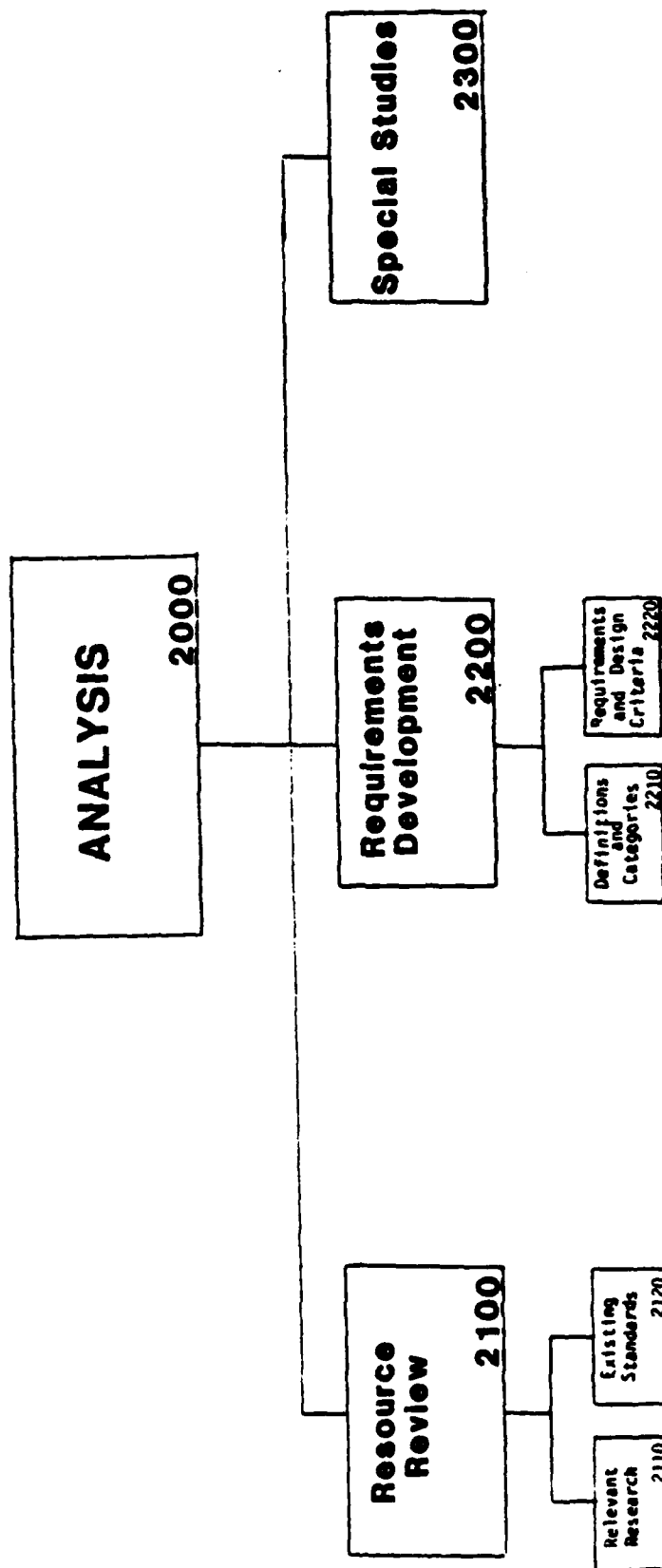


Figure 4 MBS Analysis Overview

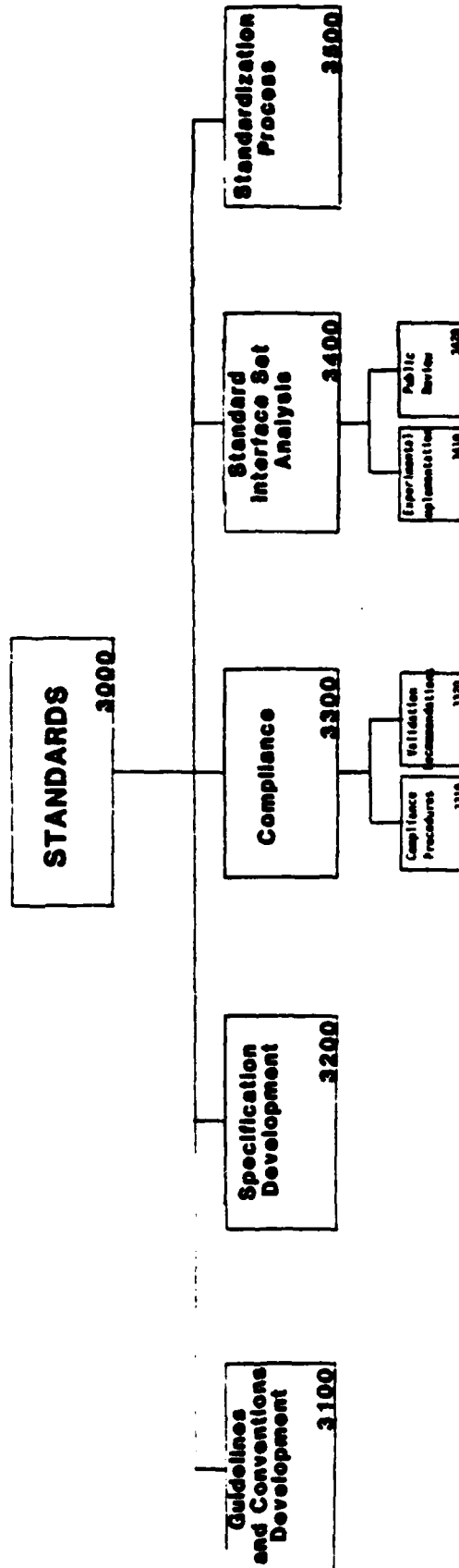


Figure 5 WBS Standards Overview

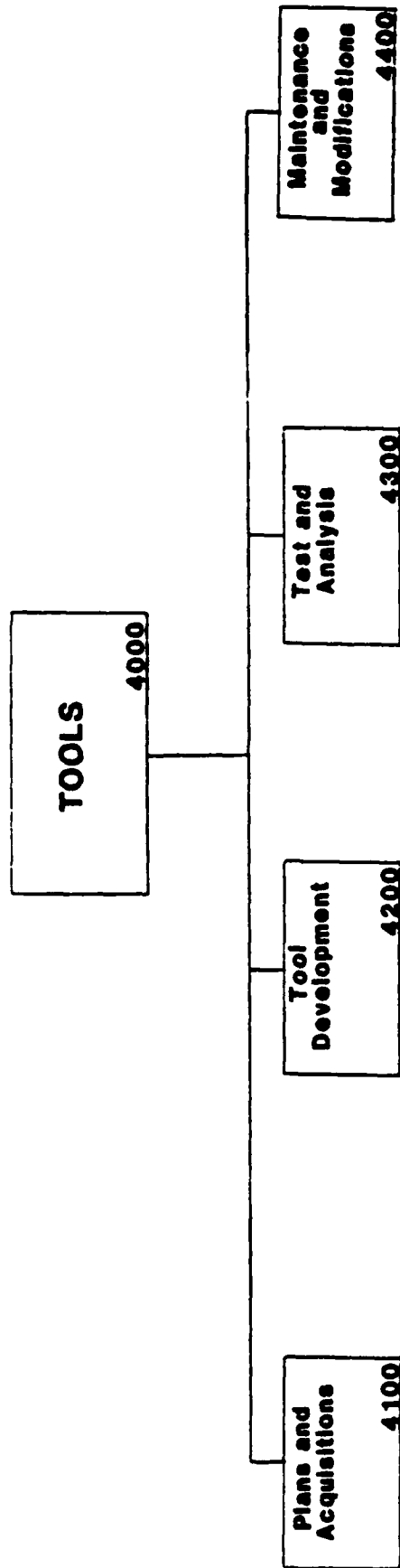


Figure 6 WBS Tools Overview

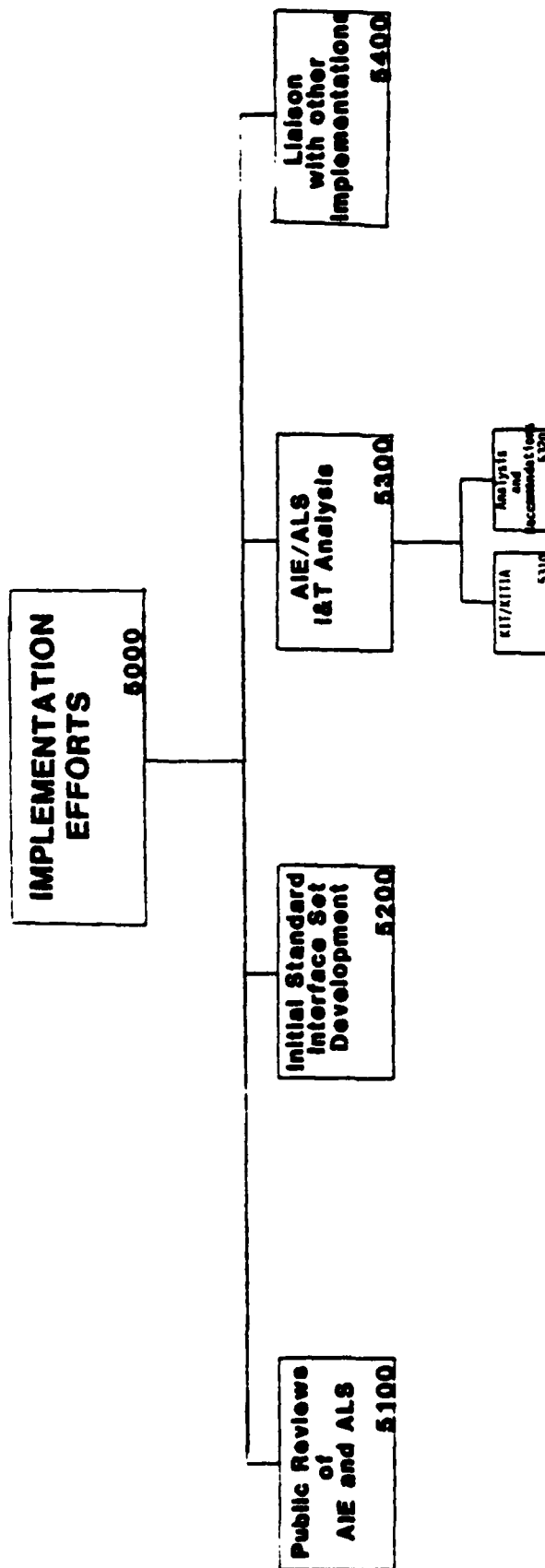


Figure 7 WBS Implementation Effort Overview

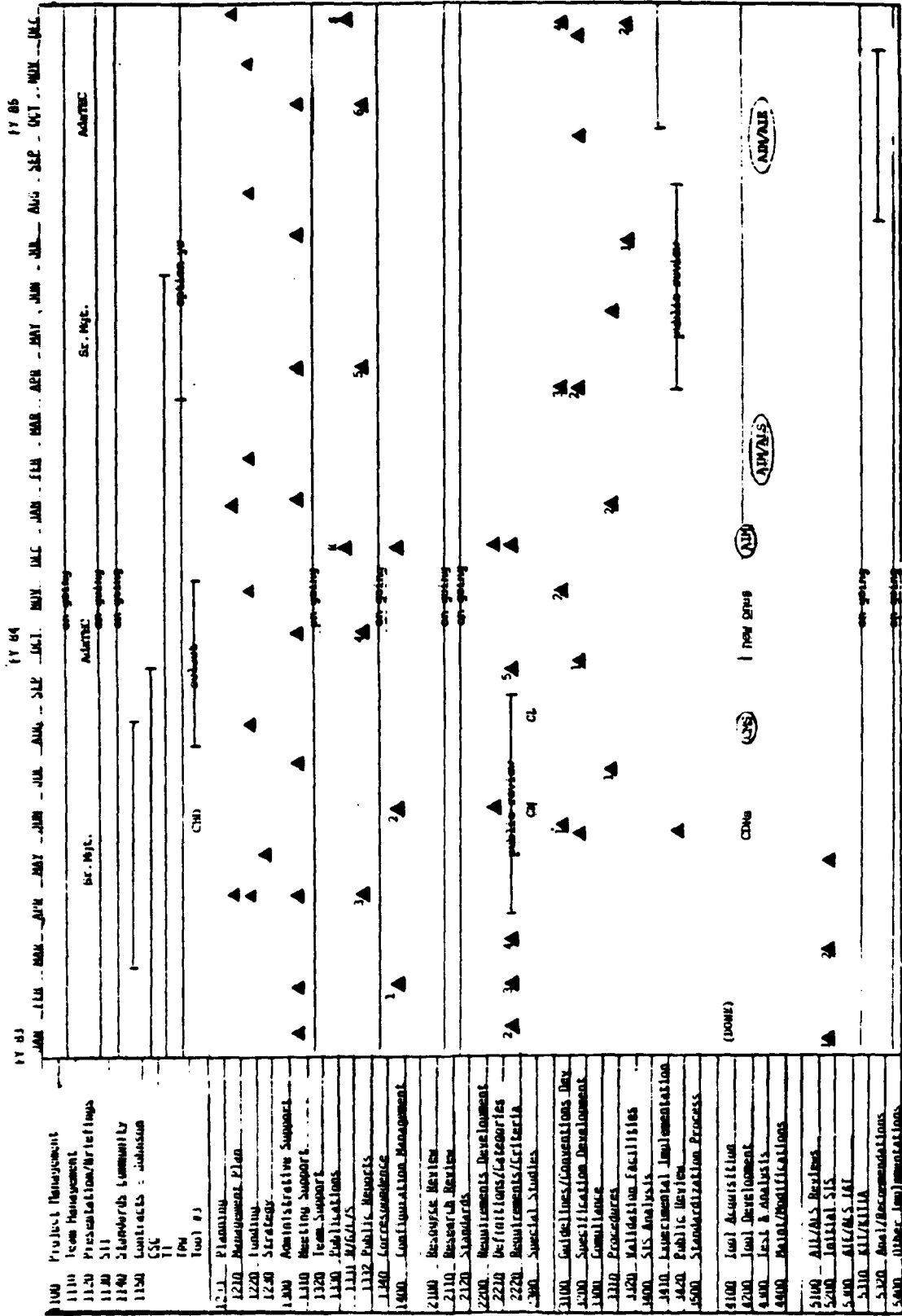


Figure 8 Schedule Summary

3.0 APSE I&T PLAN

This section shows the Work Breakdown Structure (WBS) for the I&T effort as well as the schedules and milestones for the WBS elements. Figures 2 thru 7 provide an overview for the WBS elements. Figure 8 provides a summary of the schedule.

3.1 WORK BREAKDOWN STRUCTURE

A discussion of the major elements in the WBS is presented below. Detailed task descriptions are contained in Appendix E.

1000 APSE Interoperability & Transportability (I&T) Management

This WBS element covers the general management tasks required to accomplish the APSE I&T objectives. It includes general project and team management, project planning, general meeting and team support and configuration management.

2000 APSE Interoperability & Transportability (I&T) Analysis

This WBS element covers the technical analysis tasks required to accomplish the APSE I&T objectives. It includes resources reviews, requirements development, and performance of special studies.

3000 APSE Interoperability & Transportability (I&T)

Standards

This WBS element describes the standardization tasks required to accomplish the APSE I&T objectives. It includes guidelines and conventions development, specification development, compliance and validation formulation, standard interface set analysis, and definition of the standardization process.

4000 APSE Interoperability & Transportability (I&T) Tools

This WBS element describes the development of APSE tools that support the APSE I&T objectives. It includes planning and acquisition of tools, tool development, test and analysis, and maintenance and modification of developed tools.

5000 APSE Interoperability & Transportability (I&T)

Integration with Implementation Efforts

This WBS element describes the tasks affecting various APSE development efforts required to support the APSE I&T objectives. It includes public reviews of the AIE and ALS, development of an initial Standard Interface Set, I&T analysis of AIE and ALS, and liaison with other implementations.

4.0 PROVISIONS FOR SPECIAL NEEDS

This APSE I&T Plan emphasizes the development of requirements, conventions and standards. It is unusual in that it is written for a programming language support environment that is in the development state. At this point in development it is essential for the KITEC to provide an I&T forum and act as a focal point for the Ada community, APSE developers and the DoD. This will provide broad input to the KIT from which a complete, realistic set of I&T requirements, guidelines, conventions and standards will be developed that respond to ongoing APSE development and long term APSE needs.

Normally to achieve APSE I&T the APSE itself would be written in Ada. However, STONEMAN recognizes that "in cases where there is a large current investment in software projects, written originally in other languages", provisions and guidelines must be developed that account for cost effective transitions to Ada environments. In the development of APSE I&T requirements, conventions, and standards the KITEC should provide cost benefit analysis with respect to their recommendations and decisions concerning implementation.

During the initial phase of carrying out this APSE I&T plan the KITEC will be studying and contributing to the I&T aspects of APSE developments by the Army and Air Force (ALS and AIE). When the Navy begins its development of an APSE the KITEC will also concern itself with the I&T aspects of its design. The KITEC will develop requirements, conventions, and standards that can be used for validation testing. In addition APSE development by the private sector and international development should be addressed. Criteria for validation testing for all APSE development efforts should be established. In the future a central agent can perform I&T validation testing on each APSE. The model for a strong central validation is the Ada Compiler Validation Facility.

APPENDIX A

GLOSSARY

GLOSSARY OF TERMS

AIE	Ada Integrated Environment
AJPO	Ada Joint Program Office
ALS	Ada Language System
ANSI	American National Standards Institute
APSE	Ada Programming Support Environment
DIANA	Descriptive Intermediate Attributed Notation for Ada
DoD	Department of Defense
ECS	Embedded Computer System
FCDSSA	Fleet Combat Direction System Support Activity
GCS	Guidelines, Conventions and Standards
HULWG	High Order Language Working Group
IOC	Initial Operational Capabilities
ISO	International Standards Organization
I&T	Interoperability and Transportability
JCL	Job Control Language
JHUAPL	John Hopkins University Applied Physics Laboratory
JUG	JOVIAL and Ada Users Group
KAPSE	Kernel Ada Programming Support Environment
KIT	KAPSE Interface Team
KITIA	KAPSE Interface Team Industry and Academia
KITEC	KAPSE Interface Team Executive Committee
MAPSE	Minimal Ada Programming Support Environment
MOA	Memorandum of Agreement
NAC	Naval Avionics Center
NADC	Naval Air Development Center
NAVELEX	Naval Electronic Systems Command
NAVSEA	Naval Sea Systems Command
NOSC	Naval Ocean Systems Center
NRL	Naval Research Laboratory
NSWC	Naval Surface Weapons Center
NUSC	Naval Underwater Systems Center
RFP	Request For Proposal
WBS	Work Breakdown Structure

APPENDIX B
AIE DOCUMENTS

APPLICABLE DOCUMENTS

The following documents are important sources of information relevant to the KIT effort. While the list does not represent a comprehensive bibliography on the subject of standardization, interoperability, and transportability it does constitute information sources essential to the project.

AIE Documents

- AIE Computer Program Development Specification - Part 1, CSC, March 1981
- AIE Computer Program Development Specification - Part 2, CSC, March 1981
- AIE System Specification, CSC, March 1981
- Computer Program Development Specification for Ada Integrated Environment: Ada Compiler Phases - Type B5, Intermetrics, March 1981
- AIE Design Rationale Technical Report, Intermetrics, March 1981
- Ada Software Environment Computer Program Development Specification, Texas Instruments, March 1981
- AIE Interim Technical Report, CSC, March 1981
- Ada System Specification for Integrated Environment - Type A, Intermetrics, March 1981
- Ada Integrated Environment System Specification, Texas Instruments, March 1981
- Technical Report: Designs of the Ada Integrated Environment, Texas Instruments, March 1981

APPENDIX C

ALS DOCUMENTS

APPLICABLE DOCUMENTS

The following documents are important sources of information relevant to the KIT effort. While the list does not represent a comprehensive bibliography on the subject of standardization, interoperability, and transportability it does constitute information sources essential to the project.

ALS Documents

- ALS Specification, Volume 1, SofTech, August 1982
- ALS Specification, Volume 2, SofTech, August 1982
- ALS Bare VAX-11/780 Runtime Support Library - B5 Specification, SofTech, February 1982
- ALS Vax-11/780 VAX/VMS Runtime Support Library - B5 Specification, SofTech, February 1982
- ALS KAPSE - B5 Specification, SofTech, February 1982
- ALS VAX-11/780 Linker - B5 Specification, SofTech, February 1982
- ALS VAX-11/780 Code Generator - B5 Specification, SofTech, January 1982
- ALS PDP 11/70 UNIX Code Generator, SofTech, February 1982
- ALS PDP 11/70 Assembler, SofTech, February 1982
- ALS VAX-11/780 Assembler - B5 Specification, SofTech, January 1982
- ALS PDP-11/70 UNIX Linker - B5 Specification, SofTech, February 1982
- ALS Bare VAX-11/780 Loader - B 5 Specification, SofTech, February 1982
- ALS ROLM 1666 Loader - B5 Specification, SofTech, August 1981
- ALS ROLM 1602B Loader - B5 Specification, SofTech, August 1981
- ALS ROLM 1602B Code Generator, SofTech, February 1982
- ALS ROLM 1666 Code Generator, SofTech, February 1982
- MCF Code Generator, SofTech, January 1982
- ALS ROLM 1602B Runtime Support Library - B5 Specification, SofTech, August 1981
- ALS ROLM 1602B Assembler - B5 Specification, SofTech, July 1981
- ALS ROLM Linker - B5 Specification, SofTech, June 1981
- ALS ROLM 1666 Runtime Support Library - B5 Specification, SofTech, February 1982
- ALS Compiler Machine Independent Section SofTech, February 1982
- ALS Data Base Manager, SofTech, February 1982
- ALS Command Language Processor - B5 Specification, SofTech, February 1982
- ALS VAX/VMS Symbolic Debugger, SofTech, January 1982

- ALS VAX/VMS Frequency Checker, SofTech, March 1982
- ALS VAX/VMS Timing Analyzer, SofTech, March 1982
- ALS Configuration Management Tools, SofTech, March 1982
- ALS File Administrator, SofTech, January 1982
- ALS Display Tools, SofTech, February 1982

APPENDIX D
OTHER DOCUMENTS

Other Documents

- Requirements For High Order Computer Programming Languages:
STEELMAN, DoD, June 1978
- Requirements for Ada Programming Support Environments, STONEMAN,
DoD, February 1980
- Interface Analysis of the Ada Integrated Environment and the Ada
Language System, J.M. Foidl, TRW, October 1982.
- Kernel Ada Programming Support Environment (KAPSE) Interface Team:
Public Report, Volume I, Naval Ocean Systems Center, Technical
Document 509, 1 April 1982.

APPENDIX E

WBS

WORK BREAKDOWN STRUCTURE

A discussion of each element in the WBS is presented in the following.

1000 APSE Interoperability & Transportability (I&T) Management

This WBS element covers the general management tasks required to accomplish the APSE I&T objectives. It includes general project and team management, project planning, general meeting and team support and configuration management.

1110 APSE I&T Team Management

This WBS element covers assembly of the original teams, addition of new members, establishment of working groups, general team coordination, task assignments, issue resolution and functioning as chairperson at team meetings.

1120 APSE I&T Presentations and Briefings

This WBS element covers preparation and presentation of materials for delivery to senior Government management, symposia, and professional conferences such as AdaTEC.

1130 APSE I&T Coordination with Software Technology Initiative

This WBS element covers coordination between the team and the governments Software Technology Initiative in areas of mutual interest such as interoperability and transportability of tools.

1140 APSE I&T Coordination with Standards Community

This WBS element covers interaction with various Standards organizations to identify the current procedures and methodologies for submission of team products for standardization.

1150 APSE I&T Contracts

This WBS element covers preparation and monitoring of various contract in support of the team goals. It includes preparation of task statements, review of progress, receipt and distribution of deliverables and coordination of results with ongoing team work.

1200 APSE I&T Planning

This WBS element provides for the planning necessary to follow through and complete the APST I&T program. It further provides for the updating of the APSE I&T plan on a yearly basis.

1210 APSE I&T Management Plan

This WBS element covers the preparation, review, and publication of the APSE Interoperability & Transportability Management Plan. The Plan is reviewed and published on a yearly basis to reflect the future direction of the team.

1220 APSE I&T Funding Allocation

This WBS element covers the maintenance and management of government funds for NOSC and contractor efforts.

1230 APSE I&T Strategy

This WBS element covers the planning and documentation of the overall strategy to be implemented in achievement of the team goals. It also includes the incorporation of the strategy into budget resources and team planning.

1300 APSE I&T Administrative Support

This WBS element provides the administrative support necessary in the implementation of the APSE I&T program.

1310 APSE I&T Meeting Support

This WBS element provides for the technical support required in planning, preparing, conducting and reporting on formal APSE I&T meetings. It includes preparation of agendas, discussion copies of papers, attendees lists, general meeting arrangements and meeting minutes.

1320 APSE I&T Team Support

This WBS element provides for the msintenance, storage and updating of all documentation and data in the APSE I&T program. It further provides for the distribution of all data in the APSE I&T program including maintenance of team address lists and maintenance of the ARPANET team accounts such as KIT-INFORMATION and KIT-COMMENTS.

1330 APSE I&T Publications

This WBS element provides for the publication and distribution of APSE I&T documents.

1331 APSE I&T Requirements, Guidelines, Conventions and Standards

This WBS element provides for the generation of draft version through

final version of Requirements, Guidelines, Conventions and Standards documents and their introduction into the formal publication process.

1332 APSE I&T Public Reports

This WBS element provides for the collection, preparation, and distribution of material for the KAPSE Interface Team Public Report. It also includes notification of availability to cognizant organizations and personnel and maintenance of notification lists.

1340 APSE I&T Correspondence

This WBS element provides for the correspondence via ARPANET and other communication means including physical devices and required facilities to support timely and effective team communication.

1400 APSE I&T Configuration Management

This WBS element provides for the Configuration Management of all APSE I&T documents generated and all tools developed in the APSE I&T program.

2000 APSE Interoperability & Transportability (I&T) Analysis

This WBS element covers the technical analysis tasks required to accomplish the APSE I&T objectives. It includes resources reviews, requirements development, and performance of special studies.

2100 Resource Reviews

This WBS element provides for the review of literature and documentation applicable to APSE IT requirements. Such literature and documentation will include subjects such as APSE requirements, specifications, conventions and guidelines.

2110 Relevant Research

This WBS element provides for identification of research areas relevant to APSE I&T.

2120 Existing Standards

This WBS element provides for examination and consideration of existing standards in identification of relevant areas for APSE I&T.

2200 APSE I&T Requirements Development

This WBS element provides for the identification, development and documentation of requirements for APSE I&T.

2210 APSE I&T Definitions and Categories

This WBS element provides for identification of APSE I&T related definitions and functional categories for further examination and utilization. It further provides for organization and documentation of specific categories into KAPSE Interface Worksheets for APSE I&T application.

2220 APSE I&T Requirements and Design Criteria

This WBS element provides for the development of requirements for APSE I&T. This WBS element provides for the analysis of APSE I&T Requirements developed under WBS element 2200. This analysis will be conducted to determine completeness, traceability, testability, consistency and feasibility. It also includes examination of other documentation such as the Operating System Command and Response Language Design Criteria and consideration of comments provided during public review of team products.

2300 Special Studies

This WBS element provides for any technical analysis or study not mentioned elsewhere. Specifically included are studies resulting in methods for assessing the risk associated with achieving levels of APSE I&T, and cost benefit analysis that will provide a quantitative means to assist in making recommendations and decisions concerning implementation. Examples of such special studies are possible STONEMAN revision, risk and cost assessments, and workshops to consider command languages or configuration management.

3000 APSE Interoperability & Transportability (I&T) Standards

This WBS element provides for definition, development, publication and maintenance of APSE I&T Standards. It further defines and documents criteria to be utilized in establishing compliance to the developed standards.

3100 APSE I&T Guidelines and Conventions Development

This WBS element provides for the development of guidelines, conventions and standards to be used to achieve I&T of APSE's.

3200 APSE I&T Specification Development

These guidelines, conventions and standards are to be developed from APSE I&T Requirements.

3300 APSE I&T Compliance

This WBS element provides for: the development of procedures to determine compliance of APSE development with APSE I&T specification of requirements, conventions and standards; the carrying out of these procedures to determine compliance including review and analysis of test reports.

3310 APSE I&T Compliance Procedures

This WBS element provides for definition, formulation, review and

This WBS element provides for definition, formulation, review and documentation of procedures to be utilized in compliance validation with APSE I&T documentation such as the requirements or standard interface set documents.

3320 APSE I&T Validation Recommendations

This WBS element provides for the identification and documentation of the recommendation process following validation of compliance with APSE I&T standards.

3400 APSE I&T Standard Interface Set Analysis

This WBS element provides for analysis of APSE I&T Standard Interface Sets through experimental environments and public reviews.

3410 Experimental Implementation

This WBS element provides for experimental implementation of Standard Interface Sets in various environments such as the AIE, ALS or other such environment implementations. It further provides for the identification and documentation of potential issue areas, successful methodologies employed, and recommendations for future implementations.

3420 Public Review

This WBS element provides for Public Review of the results of the Standard Interface Set developments. It further provides for collection, evaluation, maintenance and processing of submitted comments.

3500 APSE I&T Standardization Process

This WBS element provides for the methodology to be utilized in submission of APSE I&T products for standardization including format, control, revision, submissions and documentation.

4000 APSE Interoperability & Transportability (I&T) Tools

This WBS element describes the development of APSE tools that support the APSE I&T objectives. It includes planning and acquisition of tools, tool development, test and analysis, and maintenance and modification of developed tools. It also includes development of interface analysis reports to document the problems/issues encountered in the tool development process.

4100 APSE I&T Plans and Acquisition

The WBS element provides for the identification of objectives, criteria and requirements to be used for the selection of approximately three APSE I&T tools to be integrated into both the AIE and ALS. These tools will be used as

necessary to recommend the three specific APSE I&T Tools to be developed. It further provides for making the recommendation, and developing plans for the development and acquisition of these three tools.

4200 APSE I&T Tool Development

This WBS element provides for the development and acquisition of the three APSE I&T Tools to be integrated into the AIE and ALS. It further provides for the monitoring of the APSE I&T Tool development and participation in the APSE I&T Tool Development review process and the infusion and reporting of the results of monitoring and reviews.

4300 APSE I&T Tool Test and Analysis

This WBS element provides for the overseeing of the use of the three APSE I&T test tools in their integration into the AIE and ALS. It further provides for the development of guidelines for use of the tools and analysis of this use.

4400 APSE I&T Tool Maintenance and Modification

This WBS element provides for the maintenance of the APSE I&T Tools after they are developed and for any modification which may be required for the following reasons: to correct inadequacies in the first development; to stress test standards and conventions; and to respond to changing requirements.

5000 APSE Interoperability & Transportability (I&T) Coordination with Implementation Efforts

This WBS element describes the tasks affecting various APSE development efforts required to support the APSE I&T objectives. It includes public reviews of the AIE and ALS, development of an initial Standard Interface Set, I&T analysis of AIE and ALS, and liaison with other implementations.

5100 Public Reviews of AIE and ALS

This WBS element provides for support of public reviews of the AIE and ALS development efforts including collection, cataloging, maintenance, and distribution of comments, inclusion of issues in I&T plans, and submission of recommendations for future I&T considerations.

5200 APSE I&T Initial Standard Interface Set Development

This WBS element provides for the requirements analysis, design, development, publication, review and revision of an initial standard interface set for APSE I&T. This initial set will be directed to the AIE and ALS development efforts. It is intended that subsequent versions will be submitted for standardization.

5300 AIE and ALS I&T Analysis

This WBS element provides for the analysis of the AIE and ALS development efforts for APSE I&T. The results of this analysis will provide recommendations for the evolution of the standard interface set.

5310 KIT/KITIA Coordination

This WBS element provides for the coordination between the KIT and the KITIA to insure continuing progress in the analysis of the AIE and ALS for APSE I&T.

5320 Analysis and Recommendations

This WBS element provides for the definition of analysis areas of the AIE and ALS with respect to I&T and the submission, review, and disposition of recommendations resulting from these analyses.

5400 Liaison with Other Implementations

This WBS element provides for liaison with other programming support environment implementation, particularly APSEs, for identification of potential issue area and/or solutions for consideration in formulation of APSE I&T strategies, plans, analyses, and recommendations.

APSE INTEROPERABILITY AND TRANSPORTABILITY IMPLEMENTATION STRATEGY



JUNE 1983

Prepared By:

**KAPSE
Interface Team
for the
Ada[®] Joint Program Office**

([®] Ada is a Registered Trademark of the Department of Defense, Ada Joint Program Office)

CONTENTS

1. INTRODUCTION	1-1
2. GOALS AND CONCERNS	2-1
3. STRATEGY DECISIONS AND POLICY RECOMMENDATIONS	3-1
4. SUMMARY	4-1
5. REFERENCES	5-1
APPENDIX A — MOA	A-1
APPENDIX B — CONCERNS AND TRADE-OFFS	B-1
APPENDIX C — STRATEGY COMPONENTS	C-1

EXECUTIVE SUMMARY

This document discusses the goals of the KIT/KITIA effort and the concerns which have gone into the establishment of a strategy for achieving those goals. The resulting strategy can be summarized as follows:

1. There shall be one standard set of interfaces. This set shall be the subject of a formal standardization process within the DoD.
2. The foundation for this standard set shall be an initial set of interface areas in which the AIE and ALS are found to be compatible.
3. The standard interface set shall be incrementally developed by the KIT and KITIA, resulting in a candidate standard in CY85.
4. Conformance to the standard interface set will be confirmed and enforced by the use of a validation capability.
5. The DoD will maintain the standard set.
6. The standard set will be designed to be evolutionary, and the maintenance organization will be responsible for establishing a regular review procedure and a team of qualified reviewers.
7. Transition to the use of the standard set is an important consideration and will be the responsibility of each service. A strategy for public review is a part of the approach to transition.

SECTION 1 INTRODUCTION

1.1 PURPOSE OF THE KIT

The KAPSE Interface Team (KIT) was formed by a Memorandum of Agreement (MOA) signed by the three services and the Undersecretary of Defense (see Appendix A). Its purpose is to define a standard set of Kernel Ada Programming Support Environment (KAPSE) interfaces to which all Ada-related tools can be written, thus assuring the ability to share tools and data bases between conforming Ada Programming Support Environments (APSEs). This standard set will include inter-tool interfaces at the MAPSE (Minimal APSE) level as well as the KAPSE-level interfaces which provide basic services. It is especially important that the three DoD-sponsored APSEs— the Army's Ada Language System (ALS), the Air Force's Ada Integrated Environment (AIE) and the Navy's ALS/N - support this standard set of interfaces, thus making tri-service sharing of tools possible.

1.2 PURPOSE OF THIS DOCUMENT

The purpose of this document is to record the decisions that have been made by the KIT concerning the course of action which it intends to pursue in defining the required standard interface set. Many alternatives have been considered, and those decisions which follow have been based on necessary trade-offs.

1.3 BACKGROUND

The KIT was formed in late 1981 and held its first meeting in January, 1982. At about the same time a volunteer team consisting of representatives from industry and universities was also formed. Called the KAPSE Interface Team from Industry and Academia (KITIA), the purpose of this team is to act as a board of experts in various areas pertinent to the definition of this set of standard interfaces. This team generates ideas, contributes to documents, reviews KIT products and generally raises issues which must be considered in solving this standard interface problem. The KITIA held its first meeting in February, 1982.

One of the first issues raised by the KITIA was the question of DoD policy with regard to the APSEs which were under construction by the DoD (i.e., the Army's ALS and the Air Force's AIE). Although the stated goal of the KIT and KITIA was to define a set of standard interfaces to which all KAPSEs will conform, there was no stated DoD strategy for achieving that goal. This situation was found to be far too ambiguous and not conducive to widespread industry cooperation. This document remedies that situation by describing

the strategy which the KIT/KITIA will pursue and the DoD policy recommendations which, if adopted, will support the achievement of the stated goals.

1.4 DOCUMENT ORGANIZATION

The remainder of this document is organized into two sections. The first (section 2) discusses the goals which the KIT and KITIA are trying to achieve and the concerns which must be balanced and traded-off against one another in pursuing those goals. The next section (section 3) discusses the components which a strategy statement must cover and states the strategy decisions which have been made and how they support the goals and concerns of the effort. Appendix A reproduces the Memorandum of Agreement which led to the creation of the KIT and Appendices B and C provide additional detail and rationale concerning the choices that have been made.

1.5 ACKNOWLEDGMENTS

One of the first products of the KITIA was an informal statement of several options which the DoD could pursue. These were generated by Tim Lyons of the KITIA and covered a wide range of alternatives. This paper has been reviewed and discussed extensively by both the KIT and the KITIA, and many individual comments (most notably those by Dennis Cornhill of the KITIA and Hal Hart of the KIT) have been added to the deliberations. Out of these discussions has evolved the following statement of KIT/KITIA strategy. The KIT chairman hereby wishes to recognize the important inputs from those who have contributed to this effort and to extend to them appreciation for their hard work and persistence.

SECTION 2

GOALS AND CONCERNS

2.1 INTRODUCTION

In establishing a KIT strategy, there are a number of potentially conflicting goals and concerns which must be considered and weighed against one another before final decisions can be reached. The goals discussed below are those final objectives which are driving the entire KIT/KITIA effort, and indeed much of the Ada program as a whole. The concerns which follow are various aspects of how the job can best be accomplished. Each one alone is desirable to satisfy; taken together, however, their relative benefits and costs must be considered and compromises reached.

2.2 GOALS

2.2.1 Interoperability and Transportability

Interoperability and transportability (I&T) are the basic goal of the KIT/KITIA effort. These terms have been defined by the teams as follows:

- Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.
- Transportability of an APSE tool is the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming.

It is generally agreed that 100% I&T is not likely to be achieved and is not a realistic goal. The real goal of the KIT and KITIA is to make the sharing of tools and data bases sufficiently practical and cost-effective for sharing to become the normal mode of operation between the various agencies of the DoD as well as the industry which supports them.

It is also generally agreed that in order to accomplish I&T the teams must look beyond just the interfaces which come to be accepted as necessary parts of a KAPSE. In particular, the consensus is that MAPSE-level interfaces will be included in the standard interface set in order to achieve interoperability.

2.2.2 A Viable Standard Interface Set

It is the goal of this effort to define a standard set of interfaces for use in transporting tools and data bases between APSEs. This interface set is to be supported by the KAPSE and to provide the services required by tools in order to function properly in an APSE. Therefore, these interfaces include those required for data base manipulation, process invocation and control and inter-tool data formatting, among others.

It is not sufficient that this effort result in a standard interface set which is technically sound. In addition, this standard set must be achieved and administered in a way which is conducive to widespread cooperation and adherence. It is incumbent on the DoD to take more than just its own interests and concerns into consideration in the development of the standard set. The willingness of the DoD to do this has been repeatedly demonstrated throughout the Ada program and is made apparent in this work by the formation of the KITIA and the release of semi-annual reports for the broadest possible audience to utilize the standard set and for the body administering it to enforce its use. In the case of this effort, a viable standard interface set is one which will facilitate the rehosting of public domain tools, the moving of environments from a commercial setting to a government one and the building of improved environments on government-sponsored KAPSEs. The standard interface set which is put forward must accomplish at least the following things in order to succeed:

1. It must provide a full set of services to tool builders.
2. It must provide an interface which is standard across a wide variety of machines and operating systems (i.e., it must be machine- and operating system-independent).
3. It must be capable of evolving as new equipment, tools and facilities become available and desirable as components of an APSE.

It must also be possible to devise a test suite which will evaluate the conformance of an implementation to the standard set.

2.2.3 Reduced Cost

The ultimate goal of much of the Ada program is to reduce the high cost of software which the DoD has been experiencing for the last several years. However, there are both an investment and a maintenance cost involved in order to save money. Various aspects of these costs will be discussed in the following sections. The important point in general is that this standard interface set must promote reduction in costs in the long term while keeping the investment costs for everyone involved at a reasonable level during its development and introduction.

2.2.3 Reduced Cost

The ultimate goal of much of the Ada program is to reduce the high cost of software which the DoD has been experiencing for the last several years. However, there are both an investment and a maintenance cost involved in order to save money. Various aspects of these costs will be discussed in the following sections. The important point in general is that this standard interface set must promote reduction in costs in the long term while keeping the investment costs for everyone involved at a reasonable level during its development and introduction.

2.3 CONCERNS AND TRADE-OFFS

The concerns to be considered in pursuing a strategy are given in what follows. They have been grouped into four general areas: reasonableness, cost-effectiveness, acceptability and viability.

2.3.1 Reasonableness

Four concerns have been grouped together under the heading of "reasonableness." They are that the approach take into account available experience with current APSEs and other environments, that the approach provide for the incorporation of innovations, that the resulting standard set support a broad scope of interface areas and that the resulting standard interface set be able to take advantage of technology advances. Each of these concerns is further defined and its implications discussed in Appendix B, section 1.

2.3.2 Cost-effectiveness

Four concerns have been grouped together under the area of "cost-effectiveness." They are the principle of noninterference with the AIE and ALS on-going developments, the desire to limit the proliferation of non-standard APSEs, the anticipated ease of maintenance of and training on conforming APSEs and the desire to keep the cost of implementation, maintenance and training required by the standard set low. Each of these concerns is further defined and its implications discussed in Appendix B, section 2.

2.3.3 Acceptability

Six concerns have been grouped together under the area of acceptability. They are the desire for broad consensus on the content of the standard, ease of transition from current support capabilities to the standard set, the encouragement of communication and sharing between implementation sub-communities, wide use of the standard set, the promotion of adherence to the standard set and the ability of the community at large to anticipate the standard set and therefore cooperate with it. Each of these concerns is further defined and its implications discussed in Appendix B, section 3.

2.3.4 Viability

Seven concerns have been grouped under the heading of viability of the standard. They are that the standard set be controllable, that it promote commonality, that it be evolutionary, that it be extensible, that it achieve I&T, that it be complete and that it be based on simple, unified concepts. Each of these concerns is further defined and its implications discussed in Appendix B, section 4.

SECTION 3 STRATEGY DECISIONS AND POLICY RECOMMENDATIONS

The following sections describe the strategy and policies recommended by the KIT to the AJPO. They are discussed in terms of the components covered in Appendix C, and each component is related to the concerns and trade-offs which have been considered in reaching the decisions.

3.1 STRATEGY COMPONENTS

In order to succeed in this program, it is important that the strategy include the following components:

- the number of standard interface sets to be defined
- the foundation, or starting point, for defining the standard set(s)
- the approach to implementing the standard set(s)
- the approach to enforcing compliance with the standard set(s)
- the approach to maintenance of the standard set(s)
- the approach to evolution of the standard set(s)
- the approach to transitioning to use of the standard set(s).

The contribution made by each of these components is discussed in Appendix C; the decisions that make up the current strategy are presented in terms of these components in the following section.

3.2 THE STRATEGY AND POLICY RECOMMENDATIONS

3.2.1 Number of Standard Interface Sets

Although there are a number of concerns (see Appendix B) that argue for more than one standard interface set, they lack practicality in the long run. More than one standard set not only multiplies the effort the DoD must put into defining and maintaining the standard sets, but it also would lead to a situation not unlike that we have today, in which each service has its own language and support systems. Although such sub community lines need not be drawn along service lines, their existence anywhere would be contrary to the basic

goals of the Ada program in general and the KIT/KITIA effort in particular. One cannot achieve general I&T if there is more than one (incompatible) foundation on which it is to be based.

Therefore, the KIT recommends to the AJPO a policy that exactly one standard interface set be established which is to be utilized in all support systems for Ada-related work which does not receive a waiver. It is also recommended that this standard interface set be the subject of a standardization process which will result in the establishment of at least a DoD standard. The strongest argument for this decision lies in the concerns for non-proliferation of incompatible APSEs, ease of maintenance and training, and keeping the (long-term) cost of implementation, maintenance and training low. It will also help avoid the emergence of non-communicating sub-communities and will promote anticipation and cooperation by making the DoD approach clear. Most of all, it is the best means of obtaining a controllable standard set, it will definitely promote commonality (as long as other decisions assure that it is a good, workable standard), and it will achieve I&T better than any other alternative. This decision tends to work against the concerns of a broad range of experience, innovation, broad scope, allowance for technology advances and non-interference in the AIE and ALS, but its effect on these can be eased through the decisions made in other component areas (see below).

3.2.2 Foundation

With so little experience with APSEs available, it is not practical or cost-effective to start yet another one by defining a standard set of interfaces which bear no resemblance to any existing ones. Therefore the choices for basis for the standard set lies somehow with the existing APSEs. It is most logical for a DoD standard to turn to the DoD-sponsored APSEs, the AIE and/or ALS, taking advantage of others as often as possible. In doing this there appear to be two basic choices: either adopt one of the AIE or ALS (implying the restriction of the other) or derive something based on both of them. There are two very unattractive aspects of the former choice. First of all, it limits what little chance there is now of obtaining some real APSE experience and experimentation with innovations. Each of the AIE and ALS contracts has strengths and weaknesses not found in the other, so they complement one another in terms of the discoveries they have to offer. Secondly, it limits the ability to transition from one of them to the other; even though no major programs have used either of them as yet, there is a growing knowledge of them both and a certain investment of at least thinking about their use. In addition, the DoD must consider the comparative risk of making an early decision to use one and curtail the other; having "all the eggs in one basket" could prove devastating to the Ada program if the chosen one did not, for some reason, fulfill the needs of the community and resulted in extraordinary delays.

The second choice can be approached in more than one way. An attempt could be made to force the two implementations (i.e., the ALS and the AIE) into agreement in areas in which they are incompatible. However, this would be difficult and would be a critical violation of the concern not to interfere in the on-going developments. It would also have many of the negative effects cited above for the adoption of one and restriction of the other, particularly eliminating much of the basis of experience and risk reduction. Another approach would be to discover those areas in the ALS and the AIE interfaces where there is agreement or which are close enough that building agreement has no negative impact on either development. This has the positive effect of taking as much advantage as possible of work that has already been done. It likewise helps eliminate proliferation of incompatible APSEs, as the emerging standard interface set will presumably have much in common with the existing developments and will be made upward compatible with them wherever possible. It will also help to reduce the costs in the long term by not starting out wholly independently of what is already known. It will ease the transition from the AIE and ALS to the standard set and will help to eliminate the emergence of non-communicating sub-communities. Anticipation and cooperation will be promoted because developers will be able to perceive the direction which the DoD is taking. Finally, this approach will promote commonality, starting with the DoD itself, and has the added benefit of being more likely to produce a complete standard set, as it is unlikely that both developments have left major areas uncovered.

The strategy adopted by the KIT, therefore, will be to define the standard set of interfaces by first examining those interfaces which are common to the ALS and the AIE or which can readily be made common. In order to alleviate some of the interference with these two developments, the set of interfaces will initially be conceived as those which both existing designs can support on top of the interfaces which actually appear in their respective KAPSEs. This will allow both developments to continue as planned while supporting the emerging interface standard. Of course, it is quite likely that interface areas will be discovered in which agreement between the ALS and AIE is not possible. In such cases, the KIT and KITIA must decide on a course of action. The decision could be to adopt the interface approach of either the AIE, the ALS or some other emerging non-DoD APSE or to take an entirely different approach. The latter might especially happen in cases where the I&T requirements formulated by the KIT and KITIA dictate considerations which were not of importance to the AIE or ALS. It is also possible for the KIT and KITIA to decide to deviate from an interface decision even though the AIE and ALS agree on it. This would be most likely to occur in the situation just mentioned, where I&T requirements differ from those driving the AIE and ALS developments. Finally, it will be the responsibility of the KIT and KITIA to examine the resulting interface set for completeness and consistency and to make any changes that are dictated by such an examination. This strategy is not intended to guarantee that the final interface set will reflect an AIE/ALS foundation. It only suggests that the experience of the AIE and ALS developers will provide a reasonable starting point from which standard interfaces which meet I&T requirements can be evolved.

3.2.3 Implementation Approach

In order to capture the interest and cooperation of the Ada/defense community now, it would be wise for the DoD to establish a set of standard interfaces immediately. On the other hand, to prematurely move to a fixed standard set could be risky. Since the decision has been made to establish a single standard set, it is important that this be done in a way which addresses some of the concerns (see Appendix B) which argued against a single set.

The strategy of the KIT will be to build the standard interface set incrementally. Starting in early CY83 with the initial interface set common to the ALS and AIE, the KIT and KITIA will work on evolving this initial set into a final one which will be submitted for establishment as a standard during CY85. This pace will allow the teams to experiment with the interfaces, to consider their completeness and consistency and to gather considerable feedback from the community at large. It will be possible to evolve the standard set over a limited period of time, taking into account emerging APSE experience. This three-year process provides the best possible compromise between standardization "now" and the desire to define a "perfect" standard set.

This incremental development will be accomplished within the KIT and KITIA through the use of a small technical working group whose work is reviewed by the teams' full membership. Starting from this initial set which is common to the AIE and ALS, the working group will define first those critical interfaces which are absent from the initial set. This will be followed by examination of other sections of the initial set to consider those areas in which the initial AIE/ALS-based compromise will not be satisfactory for long-term I&T. Finally, the working group will undertake the definition of those additional interfaces which can be predicted to be of importance in future APSEs.

As with other AJPO activities, the participation of the general public will be sought. When the KIT and KITIA have defined a set of interfaces on which there is substantial agreement, this set will be published for wide-spread review by all those parties who are interested. The feedback obtained from this review will be incorporated in the set as appropriate before its finalization.

3.2.4 Enforcement

Initially the three services will most likely provide their APSEs as government-furnished material. However, in order to encourage experimentation and innovation, the DoD policy with respect to the standard is also expected to be that other implementations which claim to conform to the standard set will be considered for use. This means that the agency which oversees the standard set must be prepared to validate whether or not a particular implementation meets that standard. Such a validation capability is unknown today and will take time to develop, so it is possible that it would not be ready as soon as the standard interface set is. It is recommended that AJPO policy include the establishment of such a

capability. The means for achieving this and the nature of its application will be left to the newly-established Evaluation and Validation (E&V) team. Part of the KIT strategy will be to cooperate closely with those in charge of the E&V effort in order to achieve a viable means of determining conformance to the standard interface set.

The ability to validate the APSEs built by others is attractive from the viewpoint of several of the concerns. Its strongest asset is that it will leave room for contractors and others to innovate and bring new technology advances to APSEs. It will also ease the maintenance burden, as proposed changes to the standard interface set can be checked for their impact on compliance with the standard. It certainly will promote adherence to the standard set. It will make the standard set more controllable and will promote commonality, as the validation capability will make clear the interpretation of various portions of the standard. It will also assist the evolution of the standard set, as the validation capability will be maintained in conjunction with the standard and will always represent the most recent version. Finally, it will help to achieve I&T by providing a true test of standard compliance for both tools and KAPSEs.

3.2.5 Maintenance

The standard interface set will be maintained by the DoD. This maintenance will include the correction and disambiguation of interface features and documentation as well as the evolution of the interfaces over time. It will not be the responsibility of the KIT itself to serve as the maintenance organization, but one will be set up by the AJPO. The responsibility of the KIT with respect to maintenance will be to create a standard set that takes maintainability into account. The KIT will also document all the ideas it develops which will affect maintenance and see that they are made available to the organization which has responsibility for maintenance of the standard.

3.2.6 Evolution

Because a static standard will soon be an obsolete standard, the standard interface set established by the KIT will evolve. This implies that it must be constructed with evolution in mind and that the agency responsible for its maintenance must have the expertise to deal with evolution. In order to maintain I&T, the evolution must be gradual and must adhere to the same basic principles upon which the initial interface set is based.

The decision to make the standard set an evolutionary one clearly meets the arguments in favor of acquiring a broad range of APSE experience, of innovation, of providing a broad scope of interfaces and of taking advantage of technical advances. It also helps to relieve interference with the AIE and ALS in the near term. It allows for the building of a broad consensus in favor of the features of the standard set and eases transition. It will also promote wide use and adherence, since the standard set can change to keep up

AD-A141 576

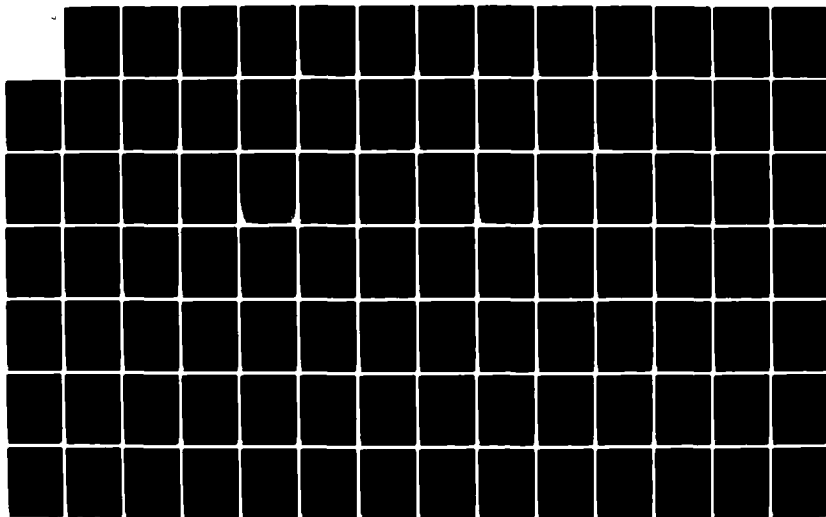
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM PUBLIC REPORT VOLUME 3(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 25 OCT 83
NOSC/TD-552-VOL-3

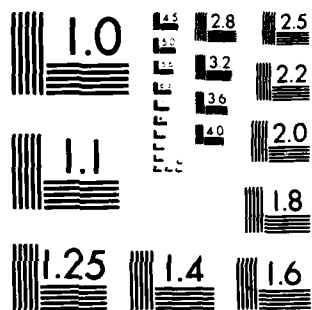
2/5

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

with new demands. It will promote commonality, since users will not be tempted to move beyond a stagnant standard set. It provides for extension and expansion of the standard set as well and helps to ensure the completeness of the set.

In order to accomplish this evolution, it is recommended that the AJPO establish a review/update schedule as well as criteria and procedures which are modelled after that used by ANSI to keep its standards current. In this model, a regular schedule for review of the standard for currency is established. Generally it will be required that changes are compatible with previous versions of the standard. Further recommendations for guidelines for maintenance and evolution will be addressed by the KIT at a later date.

3.2.7 Transition

All three services have announced plans for their movement to Ada, and all three services are implementing APSEs which will meet their unique needs. Policy concerning movement of the three services to the interface standard established by the KIT and KITIA will have to take into account all of the concerns discussed above as well as others. A careful approach to transition will help limit the proliferation of incompatible APSEs and will ease maintenance and training burdens as well as costs. It will help eliminate non-communicating sub-communities if everyone's needs are considered. It will promote wide use and adherence as long as it is feasible for all groups to move to the standard set and to keep up with it. A careful transition strategy will promote the commonality which is the major goal of the program.

The first element in this transition is already incorporated in the KIT/KITIA plans in the form of the public reports and the solicitation of public review of the teams' results. In addition, specific organizations will be requested to provide reviews of the standard interface set during CY85. Such requests for public response assist transition to the new standard by building the public awareness of and enthusiasm for the emerging standard set. In addition, the KIT will produce recommendations for further assistance to the transition process at a later date.

SECTION 4

SUMMARY

This document has discussed the goals of the KIT/KITIA effort and the concerns which have gone into the establishment of a strategy for achieving those goals. The resulting strategy can be summarized as follows:

1. There shall be one standard set of interfaces. This set shall be the subject of a formal standardization process within the DoD.
2. The foundation for this standard set shall be an initial set of interface areas in which the AIE and ALS are found to be compatible.
3. The standard interface set shall be incrementally developed by the KIT and KITIA, resulting in a candidate standard in CY85.
4. Conformance to the standard interface set will be confirmed and enforced by the use of a validation capability.
5. The DoD will maintain the standard set.
6. The standard set will be designed to be evolutionary, and the maintenance organization will be responsible for establishing a regular review procedure and a team of qualified reviewers.
7. Transition to the use of the standard set is an important consideration and will be the responsibility of each service. A strategy for public review is a part of the approach to transition.

**SECTION 5
REFERENCES**

1. STONEMAN

APPENDIX A
MEMORANDUM OF AGREEMENT AMONG
DEPUTY UNDER SECRETARY (AM)
ASSISTANT SECRETARY OF THE ARMY (RD&A)
ASSISTANT SECRETARY OF THE NAVY (RE&S)
AND
ASSISTANT SECRETARY OF THE AIR FORCE (RD&L)

Subject: Ada Programming Support Environment (APSE) Tool Transportability

Reference: Requirements definition for Ada Programming Support Environment — STONEMAN

1. Purpose

This memorandum is to establish the procedures and working relationships within which the Army, Navy and Air Force will cooperate to converge on a set of Ada Programming Support Environment (APSE) interface standards to permit the sharing of tools and other software between DoD supported APSEs.

2. Objective

The objective of this effort is to establish the necessary interface conventions for APSE tools, users and data bases to permit the consistent introduction of new tools into the software development and maintenance environment and to permit the portability of tools among different implementations of the Kernel Ada Program Support Environment (KAPSE).

3. Background

Numerous studies have predicted that the cost of DoD software will continue to escalate in the 1980s and that the availability of qualified software personnel will be a critical factor in the development and maintenance of weapon systems. The Ada Program will make the goal of a common language within DoD a reality. The high level of cooperation among the Military Departments and agencies required to establish this program has generated a unique opportunity for the DoD to adopt modern software and management practices and to develop support tools to improve productivity.

4. Agreement

We recognize that to realize the full potential of this opportunity, the DoD must focus its limited resources, including funding and talent, on the development of an Ada Programming Support Environment (APSE) which can be shared by all three Military Departments, so that software tools may be readily transported among systems and across Service applications. The STONEMAN requirements document defines the concept of a KAPSE. We agree with the concept of standard tool interfaces to the KAPSE, and a standard for all other aspects of the KAPSE which are visible to the tools. Although it may be desirable for the DoD to support different KAPSE designs to reduce risk in the early phases of the Ada Program, the long term goal is to establish the necessary interface conventions so that multiple efforts may converge to a single set of interface standards in the 1985 time frame.

The current KAPSE designs, namely the Army supported Ada Language System and the Air Force supported Ada Integrated Environment and any other KAPSEs which the DoD may support in the future will be closely monitored by the Ada Joint Program Office (AJPO) and a joint Service evaluation team to identify and establish interface conventions. The evaluation team will be chaired by the Navy. All APSE tools procured by the DoD will adhere to these conventions. In the event that, for schedule or contractual reasons, one KAPSE design violates these conventions, or if conventions are established which are not supported by a previous design decision, that KAPSE will be evolved to conform to these conventions. This agreement will be implemented through a set of procedures developed by the AJPO and coordinated by NAVMAT, DARCOM, and AFSC.

5. Duration

The provisions of this memorandum will commence when signed and will remain in effect until formally rescinded.

(signed 4 Dec 81)

Mark Epstein
Assistant Secretary of Army
(Research, Development for
Research and Engineering
Acquisition)

(signed 19 Jan 81)

William A. Long
Deputy Under Secretary of
Defense
(Acquisition Management)

(signed 14 Dec 81)

Melvyn Paisley
Assistant Secretary of Navy
(Research, Engineering and
Systems)

(signed 5 Nov 81)

Martin Chen
Assistant Secretary of Air Force
(Research, Development and
Logistics)

APPENDIX B

CONCERNS AND TRADEOFFS

There are a number of potentially conflicting concerns which must be considered and weighed against one another before strategy decisions can be reached. The concerns which are discussed below are various aspects affecting the establishment of a standard interface set for APSEs. Each one of itself is desirable to achieve; taken together, however, their relative benefits and costs must be considered and compromises reached.

B.1 REASONABLENESS

B.1.1 Broad Range of Experience

At the time of this writing, the construction of APSEs is a technology with which the community at large has very limited experience. A few contractors are at various stages of construction; the Army and Air Force systems have both been designed and most of the ALS design has been implemented and is undergoing initial test; in addition, several efforts are underway in Europe. Since the idea of an APSE as put forward in STONEMAN [ref. 1] is largely unlike any support system which exists for any current language, the entire APSE effort is breaking new ground. Certainly the construction of KAPSEs and what interfaces they should support is the newest ground of all.

This concern leads one to conclude that the current efforts should be allowed to continue unhindered for some period of time. Much useful data could be gathered on general APSE experiments and experiences and (more importantly for the KIT and KITIA) on which KAPSE interface approaches have proven most satisfactory. Indirectly, information on requirements for I&T could also be gathered and would provide more guidance for a new set of standard interfaces.

If the time to really conduct such experimental use is not available, this concern argues for taking the greatest possible advantage of what little experience is available today. This implies close cooperation with the AIE and ALS developers as well as keeping channels open for information from other developers.

In conclusion, this concern argues for:

- not perturbing the AIE/ALS
- taking time to establish the standard.

B.1.2 Incorporation of Innovations

Because of the newness of the APSE concept, there is good reason to believe that there is much room for innovation in meeting the STONEMAN requirements. These innovations cannot be anticipated today, but it is desirable that it be possible to accommodate them in the future. Like the previous concern, this one argues for taking time to allow some of these innovations to be realized so that they could be incorporated into the standard set. Whether or not such time is available, this concern argues for a careful approach to the level at which the standard set is established. The higher the abstraction level of interface specified, the more room there will be various KAPSE implementations to incorporate new innovations.

In conclusion, this concern argues for:

- taking time to establish the standard
- a high level of abstraction in the interfaces.

B.1.3 Broad Scope

This concern refers to the breadth of interface areas which the standard set defines. This is of interest from two standpoints: that of the interfaces themselves and that of the tools which are to be ported. From the standpoint of the interfaces themselves, the broader the scope is, the more effective the standard set will be in achieving its goals. In general, a standard set which covers every contingency leaves less room for deviation and interpretation and is therefore more effective in bringing the community into compatibility. From the standpoint of the tools and data bases to be transported, the broader the scope of the interface set is, the greater will be the number of tools and data bases which can be transported reliably between conforming KAPSEs. It will be much less likely that a tool can require a critical interface which will differ between KAPSEs or not be available at all. This concern also argues for taking greater time in establishing the set of standard interfaces, allowing the teams to investigate every interface which is pertinent, in an effort to include them "all."

In conclusion, this concern argues for:

- a large standard interface set
- taking time to establish the standard.

B.1.4 Allowance for Technology Advances

It is well known that technology that could affect the implementation of a set of KAPSE interface standards is moving very rapidly. Of particular interest are advances in hardware and tool capabilities. It is important that the standard set take such future advances into account and allow for them to be incorporated in APSEs as they mature. This

concern argues for an open-ended set of interfaces which do not rely too heavily on only what is available today. For example, it is clear that there is a trend today towards distribution of development support over a network of computers of various sizes; a standard set which only would operate in the context of one large mainframe computer clearly cannot rise to meet the future challenge.

In conclusion, this concern argues for:

- an open-ended set of interfaces.

B.2 COST-EFFECTIVENESS

B.2.1 AIE/ALS Non-Interference

The AIE and ALS were under contract before the MOA was initiated or the KIT formed. Neither includes a requirement to support transportability or interoperability with the other, yet that is now the DoD goal. Any perturbations to the current AIE and ALS plans will cost the DoD more money and time which it can ill afford. Besides the cost in dollars, the longer it takes to realize the DoD APSEs, the less chance there will be to gain experience with them which can contribute to the viability of the standard set which results from this effort.

Another aspect of non-interference with the AIE and ALS centers around the ease with which these two different systems can be brought into conformance with the new standard set. This concern argues for taking maximum advantage of those features which already appear in common in these two systems or which can be supported without changing the current designs.

In conclusion, this concern argues for:

- not perturbing the AIE and ALS
- using what the AIE and ALS offer.

B.2.2 Non-Proliferation of Incompatible APSEs

This concern is basic to the Ada program. A proliferation of incompatible APSEs, each independently requiring the expenditure of time and money to implement the same tools and capabilities, is unacceptable today. It is the existence of such circumstances today which prompted the DoD move to a common programming language and a common programming support environment. This concern argues for one standard interface set which would make the proliferation of non-conforming APSEs impractical and without justification from a cost standpoint, and it argues for this standard set to be established early. It argues for a well-controlled standard set which satisfies a broad range of user needs and defuses any temptations to build "another one."

In conclusion, this concern argues for:

- a single standard interface set
- early publication of the interface set
- a well-controlled standard
- a complete interface set.

B.2.3 Ease of Maintenance and Training

The ease of maintenance and training clearly affects costs. Exactly one complete standard would necessitate the maintenance of only one for the DoD, but the problem exceeds just this consideration. There are two aspects: the ease to the DoD and the ease to the contractor community. Both of these communities are also concerned with maintenance of implementations and training on both the standard and the implementations. This concern would argue not only for one standard but also for one implementation, probably controlled and maintained by the DoD. This would present the simplest of all maintenance and training situations for both the DoD and the contractor community. It might also appear to argue for a minimal standard, under the guise that the less there is to maintain and train for, the easier it would be. However, a minimal standard would only enlarge the community-wide problems of maintenance and training, since the variations made possible by the "minimum" would also have to be maintained and trained for. Therefore, this concern argues ultimately for a complete standard.

In conclusion this concern argues for:

- a single standard interface set
- a single implementation
- a complete interface set.

B.2.4 Low Cost of Implementation, Maintenance and Training

This concern has two aspects: the short-term and the long-term. In the short-term, this concern would certainly argue for exactly one standard set and for that set to be minimal; the fewer interfaces there are, the less cost to implement. In addition, it would argue for exactly one implementation, simplifying the maintenance and training needed and therefore reducing the attendant costs.

However, the long-term aspect must also be taken into account. It involves the costs to maintain the standard, to implement APSEs which conform to it, to maintain those and to provide adequate training. It also involves the costs of generating tools which can be shared because a standard exists which can be effectively used to achieve I&T. The implications of this longer-term aspect are harder to discern. On the surface, it, like the short-term

considerations, would also argue for the immediate selection of one standard and the provision of exactly one implementation of that standard. However, since any such "final" solution which we could devise today is very likely to become obsolete very quickly, this concern might in fact argue for taking the time and expending the money now to learn some things about building APSEs in general and KAPSEs in particular. This attitude assumes that a greater cost saving would result in the long-term if the time (and money) is taken now to do the job "right."

In conclusion this concern argues for:

- a single standard interface set
- a single implementation
- a minimal interface set
- taking time to establish the standard.

B.3 ACCEPTABILITY

B.3.1 Broad Consensus

One means to acceptability of a standard is to acquire the broadest possible community consensus as to its contents. One way to achieve this is to make the standard just as similar to existing capabilities as is possible, making it familiar to the community. Another way is to carry on the decision-making process in a very public manner, soliciting input and feedback from a wide range of qualified experts and potential users.

Because of the newness of the concept of an APSE and a KAPSE, it is difficult to discern just how to make the interface standard similar to existing capabilities. However, this concern would argue for the greatest possible consistency with other existing standards that are applicable, particularly the standard for the Ada language itself. The extent to which other operating system-like interface standards and/or projects will be useful is yet to be determined, but they are being considered by the KIT and KITIA. This concern also argues for extensive public exposure of the proceedings of the KIT and KITIA and careful consideration of the feedback which this process generates. Another consideration raised by this concern is that broad consensus is more easily achieved for a small set of items than for a large one. This would argue for a minimal standard, covering only those aspects for which broad agreement is likely.

In conclusion this concern argues for:

- consistency with other standards
- extensive public exposure
- a minimal interface set.

B.3.2 Ease of Transition

Transition is an issue for both the implementors and the users of the interface standard. For both, familiarity would be a very attractive trait, although, as noted above, the extent to which this can be done in the context of a new language and a new support approach is yet to be determined. Perhaps the only groups that can be realistically considered concerning ease of transition are those few who currently have or are obtaining experience with APSEs. These would be the builders and users of the ALS and the AIE as well as those few companies who are embarking on their own implementations of APSEs. To assist these groups, this concern would argue for the standard to be as compatible as possible with the on-going ALS and AIE work; it might even argue that the standard should find its foundation in what the ALS and AIE designs have in common. As for the implementors of other APSEs, this concern argues for their greatest possible involvement in the KIT and KITIA activities and the wide-spread publicity of KIT/KITIA products and proceedings.

In conclusion this concern argues for:

- compatibility with the AIE/ALS
- involvement of other APSE developers in KIT/KITIA activities
- extensive public exposure.

B.3.3 Limited Sub-Communities

There are two types of sub-communities. One arises in an application area, and this type of sub-community is to be encouraged. It is anticipated that part of the evolution of the APSE toolset will be in the direction of application-specific tools which can then be shared by all those working in the same or a closely related application area. The other type of sub-community is that which grows up around a particular implementation of a standard. It is desirable to keep this to a minimum. The goal in this effort is to make all facilities as widely available as possible; isolation of users into implementation-specific sub-communities works against this.

This concern argues for a complete standard, as most possibilities for an implementation-specific sub-community will arise from those areas which a standard neglects to define.

In conclusion this concern argues for:

- a complete interface set.

B.3.4 Wide Use

This standard must be achieved in such a way that it encourages wide-spread use. Without that, all other concerns discussed here will have a much smaller impact and the

Ada program as a whole will not achieve its ultimate goals. Wide use implies that all of the DoD must be firmly behind the standard and must be willing and able to enforce its use on various contracts. However, wide use will best be achieved if the standard is sufficiently attractive; it is always more effective to obtain voluntary cooperation than to have to resort to coercion.

This concern argues for broad publicity of KIT and KITIA activities and careful consideration of the feedback that is so obtained. It also argues for a standard which is easy to work with, easy to implement, easy to understand and easy to build on. It should facilitate the implementation of a wide variety of tools, as well as the sharing of those tools and their associated data bases.

In conclusion this concern argues for:

- extensive public exposure
- ease of use of the interfaces
- a complete interface set.

B.3.5 Adherence

As with wide use, adherence is something to be encouraged through cooperation, even though it can be enforced as well. This concern means that implementors will not be tempted to "almost" follow the standard, adding a few quirks of their own choosing which would then endanger I&T. This concern argues for a realistic standard, one which has been checked out to some degree and in which there is reason to have confidence. It must be implementable and consistent. It must also be sufficient to support the various tools which will become important in the future growth of APSEs. Although innovation is to be encouraged, it should not be necessary to depart from the standard in a nonevolutionary way in order to achieve it.

In conclusion this concern argues for:

- a realistic (i.e., checked-out, implementable) interface set
- a complete interface set.

B.3.6 Anticipation and Cooperation

Anticipation is the ability of various DoD and contractor agencies to foresee where the standardization process is leading and so to be ready with tools and trained people when it arrives with its implementations. It also implies the ability of the community at large to evolve with the standard. This concern argues, as do so many others, for broad publicity and wide-spread community review and participation in the development of the standard interface set. It also argues for early delineation of the intended DoD policy with regard to

the development of the standard set and for the early possible publication of a strawman set of interfaces which will become the standard set during the lifetimes of the KIT and KITIA. Finally, it argues for DoD sensitivity to contractor concerns, particularly with regard to specific policies on the use of the standard set, the evolution of the standard set and the transition to the standard set.

In conclusion this concern argues for:

- extensive public exposure
- early publication of the interface set
- establishment of the DoD policy with regard to the interface set.

B.4 VIABILITY

B.4.1 Controllable

This concern means that it must be possible for a governing standards body to maintain the standard and enforce its use. At the very least, this argues that the standard must therefore be clearly specified and unambiguous. Controllability also argues for a complete standard; a minimal or otherwise weak standard leaves so much room for variations that the ability to control that which IS standard becomes meaningless. Finally, this concern argues for the ability to validate the compliance of implementations to the standard. This enhances enforceability, as does wide-spread education about the standard set and its applicability.

In conclusion this concern argues for:

- a clearly specified and unambiguous interface set
- a complete interface set
- a validation capability.

B.4.2 Promotion of Commonality

This is in some sense a restatement of the overall KIT/KITIA goal. However, the mere existence of a standard interface set is not sufficient to achieve commonality. The standard set must be widely available and accessible, both in itself and in the form of implementations. It must be reasonable, practical and applicable to a wide variety of DoD work. Clearly it must be implementable. It must be more attractive to various people to use the standard set than not, and this would be greatly enhanced if it was easier to accomplish one's business using the standard set than not. Most clearly of all, this concern argues for the existence of exactly one standard interface set which is complete enough to achieve the program's goals while not unnecessarily restricting future progress towards more sophisticated support environments.

In conclusion this concern argues for:

- wide availability
- practicality
- implementability
- a single standard interface set
- a complete interface set.

B.4.3 Evolutionary

It is a widely-held belief that we are in no position today to establish a "final" set of interfaces which will suit all our needs for support throughout the lifetime of Ada. Therefore it is desirable that the standard set be capable of evolving with time and technical advances in the state-of-the-art in software support environments. This concern argues for a minimal standard interface set, consisting only of those interfaces of which we could be sure today. If such a minimal set is not an attractive alternative for other reasons, then this concern argues for a level of abstraction in the details of the interfaces which is high enough to facilitate change and/or a generality of the interfaces which allows them to change with little impact on implementations. Perhaps most strongly this concern argues for a careful decision-making process now, so as not to unnecessarily bind those areas which are most likely to change, and for a highly-qualified governing agency for the standard, which can make well-considered decisions in favor of change.

In conclusion this concern argues for:

- a minimal interface set
- a high level of abstraction in the interfaces
- qualified government agencies
- care in construction of the interface set.

B.4.4 Extensible

While evolution mainly addresses the ability of existing interfaces to change, extensibility is concerned with the ability of the user to use the features of the interfaces to create others. Like evolution, this is made desirable by our inability to anticipate future advances. This concern argues for a knowledgeable governing agency and also for care in the construction of the standard set so as to provide extension capabilities.

In conclusion this concern argues for:

- qualified government agencies
- care in construction of the interface set.

B.4.5 Expandable

Another aspect of future change is expansion: the ability to add totally new interfaces which cannot be provided in terms of existing ones. As with evolution and extension, this is made desirable by our inability to anticipate future advances. This concern also argues for a knowledgeable government agency.

In conclusion this concern argues for:

- qualified government agencies.

B.4.6 Achieve I&T

Since this concern is also one of the goals, it seems redundant to mention it. However, the standard interface set must actually achieve the goal it set out to achieve in order to be viable. As discussed in section 2.2.2, viability demands completeness, system independence and evolution. Primarily, this concern argues for care in the construction of the interface set. Preliminary KIT analysis reveals that every KAPSE interface has a potential impact on I&T.

In conclusion this concern argues for:

- care in construction of the interface set.

B.4.7 Complete

Although several of the foregoing concerns have already been said to argue for completeness, it has its own stature as a concern. Completeness means not only coverage of all possible interfaces which will affect I&T, but it also means completeness of specification of each of the interfaces which are included. This concern argues for care in the construction of the interfaces as well as wide-spread public participation in the review of the suggested standard. It also argues for the attempt to define a validation capability for the standard (as a test of its completeness) and for attempts to "implement" the standard to ascertain its practicality and implementability.

In conclusion this concern argues for:

- care in construction of the interface set
- extensive public exposure
- a validation capability

B.4.8 Simple, Unified Concepts

Simple, unified concepts are an important goal of any design effort; in the establishment of a set of standard interfaces, it is especially important. Emphasis on this concern

will result in an interface set which displays many of the qualities which foregoing concerns have shown to be desirable, such as ease of maintenance, broad consensus, adherence and completeness, to name a few. This concern argues for care in the construction of the interface set and frequent considerations of overall consistency and clarity.

In conclusion this concern argues for:

- care in construction of the interface set.

APPENDIX C

STRATEGY COMPONENTS

C.1 Number of Standards

Some of the foregoing concerns would seem to argue that it is premature to establish a standard in the near future. The technology is changing so rapidly and so little is understood about APSEs or KAPSEs that it is dangerous to standardize too early. The argument is that we should watch and wait and learn from the ongoing implementation efforts, then, gathering that experience together, define the interface standard for the next generation of APSEs. The counter-argument to this is that we cannot afford to wait. Two implementations are already underway in the DoD and many more are likely to be produced by industry, both here and in Europe, in the next few years. Once any one of these achieves some use, it will be all the more difficult to control the situation and to make everything converge to the use of common interfaces. The longer people work to adapt distinct implementations to their needs, the more diverse the communities will become and the less likely will we be to ever achieve I&T.

A midway ground would be to establish more than one "standard," thus getting the current situation under control and paving the way for future convergence. However, judging by the experience in other related areas, the likelihood of ever achieving all the potential of I&T by starting with more than one (even if it is only two) standard is small. Once time and effort have been invested in one system, there is usually too much inertia to change.

If all of the benefits of commonality are ever to be achieved, there must eventually be a single standard interface set. To be practical, that set must be defined soon enough to capture everyone's interest and attention and to make it possible for conforming implementations to be developed by the time they are required. Even though this decision is most conducive to the goals of the Ada program, it leaves little chance for learning from the current developments. Without that experience, it behooves the developers of the standard to proceed with as much care and openness as possible.

C.2 Foundation

Given that something will be standardized, the question is what its basis will be. One possibility is that either the ALS or the AIE (or both, if more than one standard is to be allowed) should provide the foundation. To choose one of them has the advantage of choosing something that is already on its way into existence, thus speeding up the process and automatically capturing one of the few groups with any APSE experience to protect.

Since the selection of one as the standard implies the discontinuation or at least restriction of the other, it also has the advantage of saving the money which would otherwise have been spent on the other, or at least reducing it to the level of a research project instead of a full development. However, neither the ALS nor the AIE was conceived of in the context of I&T. While choosing one would solve part of the problem, it is likely that some changes would have to be made to its interfaces to accommodate all of the demands of I&T.

Another possibility is to derive a set of requirements for achieving I&T and then to proceed to define a set of interfaces which fulfills them. This could be achieved in either one of two ways. One would be to start out completely fresh, basically ignoring the current developments, or at least relying on them only for experience, not real foundation. This has the disadvantages not only of not learning from what little we have experienced with APSEs, but also of creating yet another candidate for the standard which bears little or no resemblance to the two existing implementations. This makes transition from the two to the one more difficult. The other way would be to start out with a set of interfaces on which the AIE and the ALS agree and, using that for a starting point, to build a complete interface set. This has the advantage of creating something whose relationship to the AIE and ALS is more clear, facilitating transition. It also uses what little knowledge we have already acquired in building APSEs, assuming that if both of the implementations agree on the interface, it must have importance to a KAPSE. The possible disadvantage to this is that insufficient commonality may exist between the two systems, so that very little of value can be so defined. Then it will be up to the developers of the standard to augment that which is common with other interfaces in order to construct a complete set. Depending on how much can be found to be in common, this situation approaches the first way, since so many of the interfaces might be new and not bear any resemblance to either the AIE or the ALS.

C.3 Implementation Approach

This component may also be labeled "timeliness." It is concerned with the approach taken to constructing the interface set in terms of how soon the standard will be ready. One possibility is that no standard will be immediately formulated; instead, as discussed under the "number of standards" component, the teams could watch and learn from the ongoing developments and then proceed to define a standard in the late 1980's. Assuming that something is needed sooner than that in order to avert the chaos of commitments to a wide variety of implementations, the alternative of immediate standardization could be pursued. This could be achieved either by adopting either the ALS or the AIE (or both, as discussed above) or by immediate construction of a third independent set of interfaces. Such a set, in either case, could be established by the end of CY83 and be available then for use. This has the advantage of capturing the community immediately, but runs the risk of doing so with a set of interfaces which are not carefully considered or practical.

A compromise alternative would be to start with an initial set of interfaces soon. These could be derived from either the AIE, the ALS, some combination of them, or independently. Then two more years could be spent in two activities: enlarging the interface set incrementally as new areas are considered and testing out the interface set by doing partial implementations and other experiments. This would have the advantage of enough time to gather wide public feedback on the way the interface set was developing, while capturing everyone's attention and diminishing the likelihood that they would be tempted to strike out on their own without a standard.

C.4 Enforcement

One means of enforcing the standard is for the DoD to accept only that work done using a government-furnished implementation. Then all responsibility for the adherence of the implementation to the standard lies with the government. As long as some evidence can be examined to determine that the work was indeed accomplished using the government furnished support environment, no further enforcement would be required, other than the normal ones of informing everyone in the DoD of the requirement and considering the granting of waivers. However, this also leaves the government with the responsibility for all innovation and enhancements for APSEs. It would be desirable to take advantage of the many contributions which the contractor community is likely to make. This implies that it would be desirable to allow the contractors to implement *their own APSEs*.

This alternative places the government in a somewhat different position. Now, rather than guaranteeing that the DoD implementation is sound, the DoD must be prepared to validate that APSEs submitted to it by various contractors meet the standard. Without such an ability to validate, the standard would be largely unenforceable; each contractor would be able to claim compliance, but minor variations are likely to exist due to varying interpretations, and major variations could also exist because the contractor felt compelled to improve upon the standard. Only through rigorous validation could the DoD hope to control the standard. The model which could be chosen here might resemble the approach being taken to the validation of Ada compilers.

C.5 Maintenance

At the very least, the DoD must be responsible for maintaining the standard itself; depending on other decisions, it could also be responsible for maintaining one or more implementations of that standard. Despite the fact that some maintenance responsibility is a foregone conclusion, there are decisions to be made regarding such things as who will be the responsible agency, how Configuration Management will be performed and how changes will be distributed.

C.6 Evolution

It is not necessary that the standard evolve. It could be declared to be complete and final when it is first issued and only implementations be allowed to evolve as long as that evolution is not in conflict with the standard. On the other hand, it is expected that a large number of advances and new ideas will emerge in the next few years, not only with respect to APSEs but also with respect to technology in general. It would seem appropriate to allow the standard to change with these new discoveries. If it does not, it runs the risk of becoming obsolete and unusable.

If the standard is to evolve, a strategy must be established regarding how that evolution is to take place and in what ways the standard will be allowed to change. In order to maintain I&T, the evolution must be carefully controlled.

C.7 Transition

Transition concerns the means of moving from today's world (including the early APSE implementations, which will probably not be written to the standard) to the world of widespread use of standard-conforming APSEs and I&T. Transition is not a simple matter of dropping the way one does business one day and taking up a new way the next. It must be planned and carefully pursued so as not to disrupt the functioning of the DoD or its contractors. Strategies in this area must also be coordinated with general policies concerning Ada itself.

KAPSE FILE STRUCTURE

Rudy Krutar
Naval Research Laboratory

BACKGROUND: The KAPSE Interface Team (KIT) is supposed to specify a standard KAPSE interface to support interoperability and transportability of APSE software. See appendices A and B for diagrams illustrating relations among Ada tools and Ada Programming Support Environments (APSEs).

A KIT working group has proposed a KAPSE file structure based on UNIX. It specifies a hierarchical file structure with files designated by full or relative pathnames as in UNIX, except that components of a pathname are delimited by .'s instead of /'s, and a pathname beginning with two delimiters has a special meaning. Pathnames are strings and require string operations.

PROBLEM: A frequent transportability problem is that some programs designate auxiliary files by full pathnames, but the receiving host has a site-dependent collection of directories; so the full pathname is not valid on the receiving host.

Another problem with the proposed file structure is that common operations on pathnames must be programmed in terms of string manipulations. Such programming is surely implementation detail, and a violation of widely accepted principles of information hiding.

SOLUTION: The standard KAPSE file structure is best specified as a package of operations on file names, which in turn are specified as a standard Ada type (say, the 'Knode'). Knode operations should be designed for convenience in developing and maintaining Ada programs that manipulate files. User convenience should be supported at a higher level — the APSE file structure.

PURPOSE: The purpose of this document is to suggest an alternative standard KAPSE file structure to better support interoperability and transportability.

CONTENTS: The rest of this document discusses the following topics in some detail:

1. REQUIREMENTS: general requirements for a KAPSF file structure,
2. CRITERIA: desirable features of a KAPSF file structure,
3. ALTERNATIVES: broad approaches to file structures,

4. FUNCTIONALITY: major areas of functionality to be supported by the KAPSF file structure,
5. ATTRIBUTES: properties or fields that could be maintained for each K-node,
6. HIERARCHY: how the 'flat' K-nodes form a relational hierarchy.
7. ISSUES: a list of sticky questions about KAPSE files,
8. OPERATIONS: a list of pseudocode calling sequences (most of which can be coded in Ada) for operations on KAPSE files,
9. CONCLUSION: a summary of recommendations.

Two diagrams are attached as appendices to show relations between various tools and programming environments:

- A. APSE OVERVIEW: a diagram showing how APSE, MAPSE, and KAPSE cooperate,
- B. MAPSE TOOLS: a diagram showing relations between tools in a Minimal APSE (MAPSE), which would be supported by the KAPSE,

1. REQUIREMENTS: Any standard KAPSE file structure should satisfy the following identified requirements:

Transportability -- the degree to which an APSE tool can be installed on a different APSE without reprogramming; the tool must perform with the same functionality in both ASPSSs. Commonly used synonyms are portability and transferability.

Interoperability -- the degree to which APSEs can exchange data base objects and their relationships in usable forms without (explicit) conversion.

Habitability -- the degree to which desired APSE tools can be built in in terms a specified interface to a host without working around it; that is, how complete the interface is.

2. CRITERIA: A good standard KAPSE file structure will also satisfy the following design criteria:

Convenience -- Operations that are frequently coded should be easily coded.

Efficiency -- Operations that are frequently executed should be quickly executed.

Thrift -- Structures that are frequently used should not waste valuable resources.

3. ALTERNATIVES: A file structure may be flat (as in UNIX I-

nodes), hierarchical (as in UNIX directories), relational (as supported by UNIX wild cards), or distributed (as in GRAPEVINE at Xerox PARC).

A flat file structure is the simplest and easiest to develop. Many systems have a high-level file structure built on top of a flat one, and a relational data base built on top of that.

A hierarchical file structure is common and well understood. However, hierarchical file structures are often misused by appropriating the top levels for device selection and accounting, both of which are site-dependent.

A relational file structure could be best for supporting transportability by use of relative, usage-oriented file names.

A distributed file structure would have files on many different devices and hosts, better supporting interoperability.

4. FUNCTIONALITY: The following functional areas contribute toward satisfying various requirements and design criteria:

Security ---- The KAPSE file structure should be robust, protecting projects and programs from both innocent and malicious damage.

Database Management -- The KAPSE file structure shall maintain the database required for development and maintenance of project software so that that database pertaining to the project is transportable to other KAPSE hosts.

Configuration Management -- The MAPSE or KAPSE file structure should make development and maintenance of multi-host distributed software possible and relatively easy.

Accounting -- The KAPSE file structure should provide mechanisms for collecting management information, such as cost accounting and activity monitoring.

5. ATTRIBUTES: Let a KAPSE file be designated by a 64-bit key, which shall be called a 'K-node'. That key may be stashed anywhere in any form and reconstituted later. The KAPSE shall not attempt to track all copies of such keys.

Some portion of the 64-bit key will be used as an index in internal KAPSE and MAPSE tables. The remaining bits will serve as a check that the key is a valid K-node; 64 bits is enough to ensure that no K-node need ever be reused. Some file attributes and their domains must be standardized for Interoperability and transportability. Desirable file attributes include the following, which are grouped by functionality:

For Security:

KNODF ----- the 64-bit key; access via any other key is invalid; this attribute is changed when a K-node is deleted.

ACCESS ---- who can do what to this KAPSE file: READ, WRITE, MODIFY, APPEND, EXECUTE, INTERPRET by a given program, DELETE.

CLASSIF --- the military security classification of the KAPSE file; system-high operation will be necessary; files with a higher security classification may not be created. The KAPSE file structure shall not allow classified data to be passed through it to less classified hosts.

PASSWORD -- an encrypted version of the access password for this file, if any.

CRYPTO ---- a reference to the public key under which the file is encrypted, presumably enough for an authorized person to find the private key that decodes the file.

CHECKSUM -- a standard redundancy check on the content of a file for quickly deciding that the file has been corrupted or differs from a file on another host.

For Data Base Management:

WHERE ----- host-dependent descriptor of where the content of the file is kept (host, device, pathname, component, ...);. this may be a rule for generating the content from other K-nodes.

TYPE ----- a descriptor of the Ada type of each record in the file; some standard record types, such as character and integer, should be supported uniformly for interoperability in all KAPSE file structures. This attribute records the representation mode of a file; what it is is recorded as the LANGUAGE attribute.

SYNCH ----- synchronization status of the K-node with respect to concurrent processes accessing the K-node in read, write, append, and delete modes.

REFS ----- how many copies of the key have been distributed; K-nodes with no outstanding references may be deleted. Privately copied keys should not be returned, until all copies are obsolete.

EXPIRES --- when the K-node can be deleted peremptorily; outstanding references become invalid; no key will ever be reused.

MIGRATE --- a rule to be applied when the file expires: simplify it, move it to a cheaper medium, delete it, etc.

PURGE ----- a rule for removing unneeded versions of the file; versions that exist on backup media should be retained as long as the backup version is available.

For Configuration Management:

REVISED --- when the content of the file was last modified;

SOURCES --- a table mapping names into K-nodes from which the content of the file can be recreated; if any source file is newer than the file to be read, then the file to be read should be recreated. K-nodes without sources designate primary input files. A generated file is thereby a file directory, which models a configuration. A long document should be composed from sections, and sections from pages so that listings can be updated by change pages, but edited as though it were a single file.

REFRESH --- a rule for recreating the content of the file from its sources.

NEEDS ----- a mapping from K-nodes to expected revision date-times; this entry is updated whenever the designated file uses a new feature of another file or stops supporting a feature used by another file; these mappings will be combined mechanically when components are combined into subsystems, verifying that the revisions combined are intended to be combinable.

LANGUAGE -- the language of the content of the file: Ada, Diana, Object, Linda, English, ...; a language processor will check whether its input file is in a sublanguage of the expected language.

For Accounting:

OWNER ----- who is responsible for the file; perhaps the main K-node for which this one is a source, so that each K-node then has a standard full pathname.

ACCOUNT --- which account is to be billed for the file.

SIZE ----- length of the file in bytes.

COST ----- billing rate for the file based on where it is.

6. HIERARCHY: The proposed file structure is hierarchical in that each K-node may be viewed as a directory of its sources.

A command library would be a K-node with named sources referring to executable programs. A search path would be a K-node with numbered sources referring to command libraries.

7. ISSUES: The following issues need to be resolved for any standard KAPSE file structure:

- a. Does a KAPSE file refer to a particular file content or to the current version of an evolving file? (The latter is preferred.)
- b. If the previous version of a file is generated from the current version and a list of differences, then how can the content of the current version be changed again without affecting the regeneration of the previous version. (Use a 'current version' K-node composed from extant previous versions.)
- c. Which file attributes and operations should be supported directly by the KAPSE? Which should be supported at higher levels?
- d. How does a programmer get sole responsibility for changing a piece of software for a while? How does he release revisions of that software? How does that software get certified, released generally, and distributed?
- e. A sequence of revisions of a file could be represented by a K-node, which has each revision of the file as a numbered source. Is this a better approach than having a PREVIOUS attribute?
- f. How can the need for site-dependent file names be avoided? The problem is a lack of predefined places to start relative accesses from.

8. OPERATIONS: Let A, B and C be K-nodes. The following operations shall be supported (the phrases are pseudocode statements to be encoded as Ada statements):

For Database Management:

C := A-B; compare KAPSE files designated by A and B; record their differences as a (new) file of change directives that would recreate the content of A from the content of B; set the revision date of C as the most recent of the revision dates of A and B. A and B must have the same record type T; C will have a generated record type delta(T) and language 'Changes'.

A := C+B; define A as a new KAPSE file, the content of which is generated by applying the change directives in C to the content of B; use the maximum revision date. If B has record type T, then C must have record type delta(T), and A will have record type T.

For Configuration Management:

let A >= B;	indicate that A depends on a new feature of B.
let A > B;	indicate that A depends on a future feature of B.
for X in A do S1;	refresh each source of A; combine the NEEDS mappings of the sources of A to check for compatibility and to obtain a NEEDS mapping for A; if A is older than any of its sources, then refresh A from its sources, and indicate when A was revised; verify that X is of the record type of A; generate each record X in the contents of A, and execute statement S1 accordingly;
let A = F(B,C);	define A as a new K-node that is recreated from B and C by applying language processor F; list B and C as first and second sources of A; enter the range of F as the LANGUAGE of A;
A := Knode K;	convert 64-bit integer K to type Knode, as may be required if a K-node key is stored offline.
Knode K := A;	convert K-node A to a 64-bit integer K for offline storage.
for A: KAPSE do S;	generate all valid Knodes A, executing S for each.

For Hierarchical File Structure:

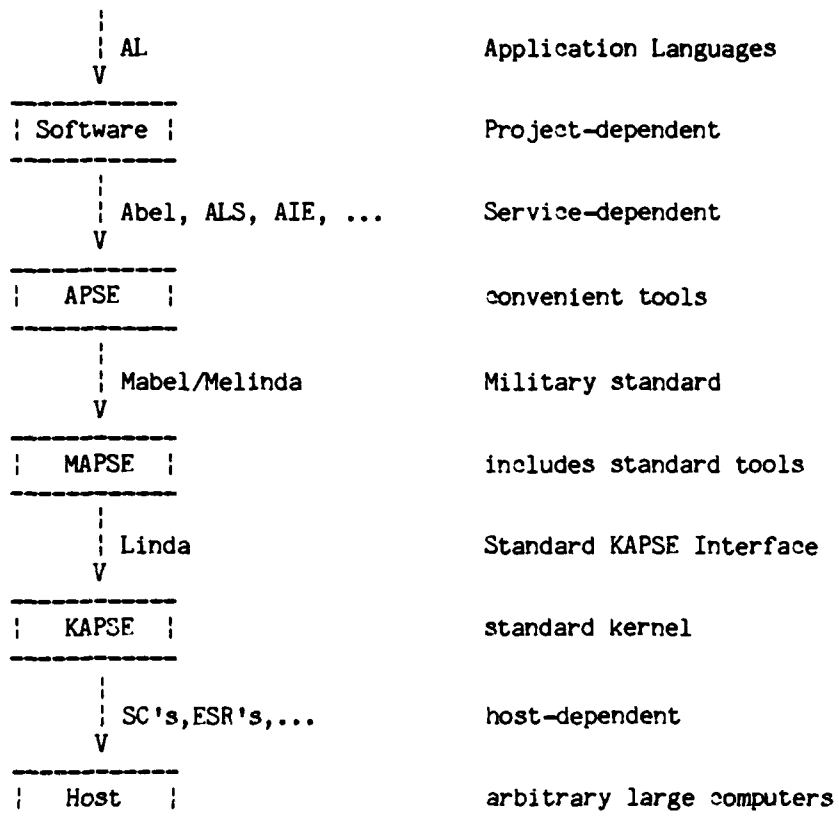
B := A/NAME;	set B to the NAMEd source file for A.
A/NAME := B;	indicate that B is the NAMEd source file for A, where NAME is a string.
A/NAME := ();	indicate that A no longer depends on a NAMEd source file.
for A/NAME=B do S;	do statement S for each source file of A with NAME and B set accordingly.
HOME_NODE	a constant naming the K-node associated with the current user; its sources are the file structures to which that user has direct access.
THIS_NODE	a constant or variable naming the current focus of attention for the user.
THAT_NODE	a constant or variable naming an alternate focus of attention for the user.

PROG_NODE	a constant naming the K-node of the currently running program.
LIST_NODE	a constant naming the file that is being regenerated by running the current program; its sources should include the program and auxiliary files used by it.
ROOT_NODE	a constant naming the K-node associated with the entire file structure.
USES(A)	returns a sequence of K-nodes that name A as a source.

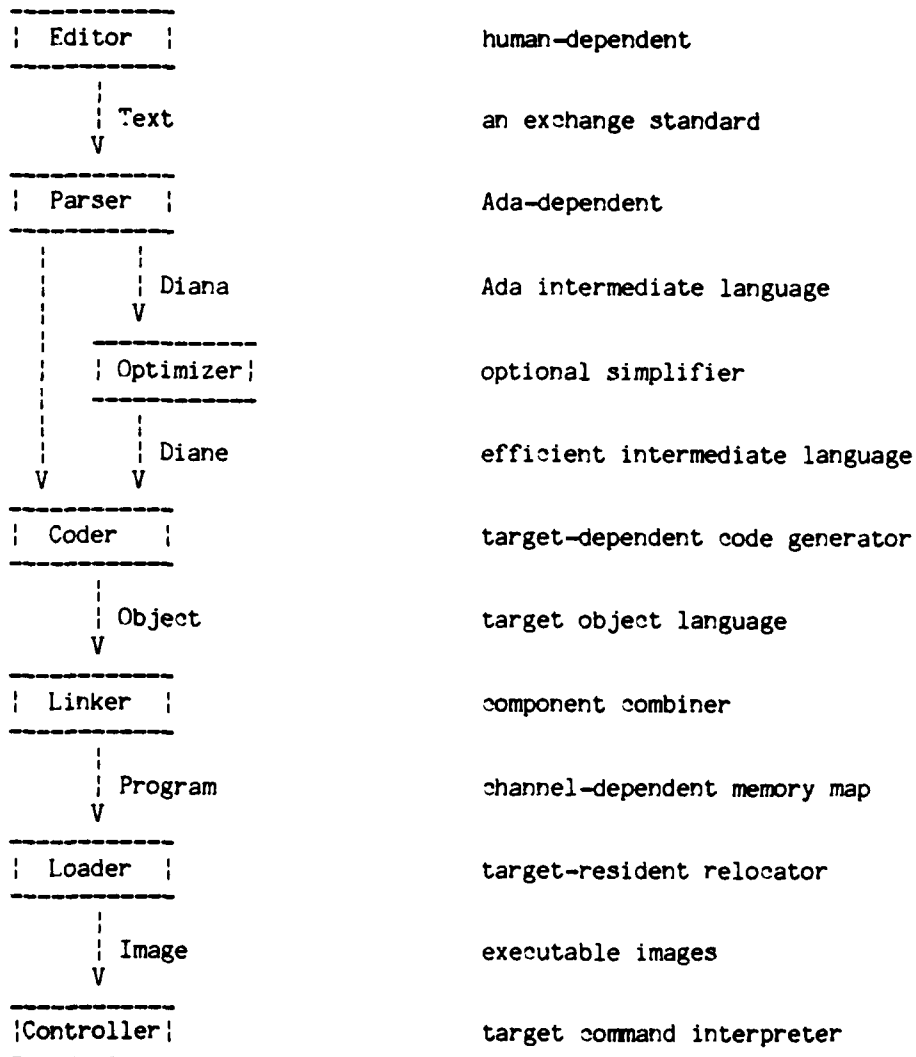
CONCLUSION: The above discussion has outlined a file structure that supports configuration management of software during both development and maintenance. A good standard configuration management tool built on such a file structure would greatly enhance the transportability, event the releasability, of developed software. A systems manager procuring such software would not have to guess how the software is put together -- its configuration is part of the file structure.

I recommend that the suggestions in this paper be seriously considered in terms of where they ought to fit in an APSE, MAPSE, and KAPSE. Those that best promote transportability and interoperability should be included in the statnard interface file structure or data base.

A. APSE OVERVIEW: The APSE, MAPSE, and KAPSE cooperate as indicated in the following diagram, wherein the arrows represent interface languages:



B. MAPSE TOOLS: The following diagram represents connections between various standard MAPSE tools, the arrows again represent interface languages:



A Virtual Terminal Specification and Rationale

Stewart French
Texas Instruments, Inc.

Abstract

The character-imaging computer terminal, consisting of a display device and keyboard, is the most widely used means of communicating with computer systems. Even now, with relatively well developed techniques for device independence, programs tend to be targeted for either specific character-imaging computer terminals or a very small subset of character-imaging computer terminals. A complete intermediate-level virtualization of a character-imaging computer terminal will promote program transportability, high level terminal abstractions, and maximum flexibility for a tool writer. If this intermediate-level virtualization conforms to an existing standard for the device characteristics of the character-imaging computer terminal (ANSI X3.64-1979), there will be many advantageous side-effects. The virtualization will closely match the functional characteristics of the conforming character imaging computer terminals, the virtualization will be more widely accepted, device independence can be promoted, and upward-compatibility of the virtualization will more closely follow the upward-compatibility of the standard.

Introduction

This document presents a specification and rationale for an intermediate-level virtualization of character-imaging computer terminals. The information presented covers six areas: an introduction to virtual terminals and device independence, a multi-class and multi-level approach to characterizing existing computer terminals; a four layer approach to terminal virtualization; an introduction to computer terminal standardization efforts; and, conclusions.

A virtual terminal is "a conceptual terminal which is defined as a standard for the purpose of uniform handling of a variety of actual terminals" [DAV79]. There are four goals that should be addressed in specifying a virtual terminal. A virtual terminal should enhance transportability of programs that perform computer/terminal interaction; provide a common interface for terminals produced by a wide variety of manufacturers, provide the tool developer with an extensive set of interactive terminal control functions; and, provide the virtualization at a level that supports many different models of computer/terminal interaction.

An Intermediate Level Virtual Terminal Rationale

The virtual terminal can be defined at many levels. The low level virtual terminal presents the tool writer with a model of the computer terminal that supports direct manipulation of the terminal hardware. An intermediate level virtual terminal presents the tool writer with a model of the computer terminal that closely resembles the functional characteristics of the terminal hardware. The highest level virtual terminal presents the tool writer with a model which hides the functional characteristics of the terminal hardware.

A virtual terminal at the low level provides a tool writer with maximum flexibility. An example of a low level model is the actual device interface presented to the tool writer from a modern operating system. But the tool writer must completely define his own model of the computer terminal with each tool he writes. With many tool writers (alas, even with the same tool writer), many different terminal models at all levels of sophistication could then exist. This would promote confusion. Also, the flexibility gained at this level allows a tool writer to indiscriminantly use facilities of one terminal that may not be available at any other terminal, even with simulation. This promotes undesirable device dependence.

A virtual terminal at a high level imposes a model on the tool writer. An example of a high level model is the concept of equating a physical terminal to a text file and using text file I/O techniques to address and control it. This model tends to hide the functional characteristics of the terminal hardware. It promotes device independence and generally provides a clean interface with the terminal. The disadvantages of this level are twofold. First, the tool writer may wish to define a different model of the computer terminal than the one with which he is presented. He would have to define the new model in terms of the original model that itself does not model the terminal hardware. Even modeling the terminal hardware is awkward. The tool writer must model the terminal hardware in terms of the existing model that hides the hardware. Second, the ability to integrate new terminal hardware into the existing model promises to be very difficult.

An intermediate level virtual terminal that presents the tool writer with a model of a functionally advanced hardware definition of a character-imaging computer terminal permits great flexibility for the tool writer while maintaining a level of abstraction. An example of an intermediate level model is an abstract representation of the functionality found in most advanced computer terminals. Since the virtual terminal models the advanced computer terminal, those functions supported in the

terminal hardware are available to the tool writer directly. Those functions not directly supported in the terminal hardware can be simulated within the virtual terminal. This will be discussed later. The tool writer could then define his own high-level abstract model in terms of this intermediate-level abstract model. Through a proper choice of the intermediate-level interface, the virtual terminal could model most of the terminals on the market today and provide upward-compatibility for new hardware.

Device Independence

Device Independence techniques.

Device independence is typically achieved by assuming some standard device and mapping existing devices into the standard. There are essentially three choices for the standard device: the simple device, the complex device, and some device of intermediate complexity.

The simple device is typically the easiest to implement and manipulate. An example of a simple device is a printing terminal. The major problem with the simple approach is that it does not address the advanced features of the computer terminals on the market. A user who purchases a computer terminal with advanced features will reasonably expect to be able to use some of them.

The intermediate complexity device has essentially the same advantages and problems. An example of an intermediate complexity device is a 2-dimensional display device with direct cursor addressability. It is reasonably easy to implement and manipulate but does not provide the support for the advanced computer terminals on the market. This level would be the most frustrating to work with. A tool writer would begin to get a

flavor of what he wanted to do but may not be able to completely "get at" the features he needs.

The complex device has an interesting set of advantages and disadvantages. An example of a complex device would be an ANSI compatible computer terminal with advanced functions such as graphic rendition (highlighting, blinking, etc), insert line, delete character, etc. The complex device would have a robust set of operations available. A tool writer would be presented with a very complete set of procedures and functions from which to choose to accomplish his goal. He would be able to take advantage of many features available on the most complex computer terminals on the market. The complex device would be the most difficult to implement and manipulate. If a user did not have a computer terminal that supported the operations defined in the complex device, it would be left to the device driver or some simulation level of software to sustain the myth that he did have such a device.

Although the complex device requires some level of simulation to achieve the advanced features on simple terminals, it appears to be the most advantageous approach. Computer terminals are now reaching the market that incorporate many advanced features. The terminal manufacturers are beginning to incorporate the concepts and functionality defined in the ANSI standard X3.64. To conform

to the ANSI standard has very worthwhile effects. It helps formulate the model as a complex device that promotes device independence in character-imaging computer terminals. The conforming computer terminals appear much the same from one manufacturer to another in terms of their operations.

Device Independence in Existing Applications. Four concepts concerning device independence that are applicable to character-imaging computer terminals are presented in this section. The concept of layered communication is derived from the field of computer networking [DAV79]. The concept of the terminal capabilities database was developed by Bell Labs for the UNIX operating system [UNIX is a registered trademark of Bell Labs] to promote computer terminal device independence [JOY81]. Two concepts are derived from the field of computer graphics: The concept of levels and classes of support [COR79], and the concept of "bundling" attributes [GKS82].

A two level approach to device independence has evolved out of necessity in the field of computer networking. A computer terminal communicates with a local controlling intelligence to perform human/computer interactions. The local intelligence then communicates with the remote host in a standard way to transfer the data the terminal user produced or requested. In this manner, regardless of the actual terminal connected to the local

intelligence, the remote host sees all terminals essentially the same way. It is up to the local intelligence to make use of the features available on the computer terminal. [DAV79]

Bell Labs' UNIX operating system provides the tool writer with a database containing information on many terminals currently on the market. Using the database, a tool can be made device independent to a certain degree. Of course, if a tool writer makes use of a facility of his terminal that is not supported on another terminal, it is up to the tool writer's software to simulate that facility or not proceed with the execution of the program. [JOY81]

The field of computer graphics probably has the most difficult task in trying to achieve device independence. There are many different kinds of graphic devices for both input and output. There are two widely accepted proposed standards for computer graphic devices: the ACM CORE standard [COR79], and the ISO Graphical Kernel System (GKS) [GKS82]. This terminal virtualization does not encompass computer graphics devices; such material is available in the references. The two important derived concepts that are applied to this virtualization are the concept of levels and classes of support from the CORE; and the concept of "bundling" attributes from the GKS.

The GKS standard has defined a method that combines graphic attributes such as color, line width, dashed representation, etc. These attribute combinations are called bundles. For example, a program which draws a line on the display could specify a bundle number of 1. This line drawn on one display device would have a particular representation (such as dotted and red). On another display device the line would perhaps have a different representation (such as solid, thin, and black). Regardless of the eventual representation the same program would execute on either graphics display device. Since the program only specifies the bundle numbers and not the actual representation, it is left to something other than the program to determine the actual representation the user would see at his graphic display device.

The CORE system has a complex set of levels for input, output, and dimension. In order for a given device to provide support for a given set of levels of input, output, and dimension, the device must implement all of the features defined in that combination of levels using any of the following: direct hardware support; hardware simulation support; or software simulation support. It must also implement no features found at higher levels (even if the hardware itself supports it directly).

The Virtual Terminal Classes and Levels

To apply the concepts presented above, the terminals are divided into different classes based on their operations and characteristics. A class identifies a set of operations and characteristics that may or may not intersect another classes' set of operations. Each class is subdivided into levels of support. Increasing levels within classes identify additional functionality of the terminals in that class. A level that is said to be above another level within the same class is a superset of the lower level.

The number of classes should be small to incorporate as many different terminals as possible into each class. This increases the amount of simulated functionality for those terminals that do not support every function in its class and increases transportability.

A terminal that appears in a particular class and level must support all of the functionality (and only the functionality) defined in that class.

There are three obvious classifications: the scroll mode terminal, the page mode terminal, and the form mode terminal. These are numbered 0, 1, and 2.

The scroll mode terminal encompasses those terminals that operate like hardcopy terminals. That is, when a carriage return (or any terminator) is typed the terminal scrolls one line upward. The functionality of this terminal class is severely limited and therefore, is the simplest of the terminal classes.

The page mode terminal encompasses those terminals that have a two-dimensional display screen that can be directly addressed and may or may not have any local intelligence (i.e. VT100, Concept-100, Visual-50). This class of terminal encompasses 75 percent of the terminals on the market today.

The form mode terminal encompasses those terminals that have form fill-in capabilities (i.e. IBM 3278). That is, the application program presents the user with a form to fill in on his display screen. The user fills in the form by interacting with the local intelligence and transmits the data to the host through some special keystroke(s) (i.e. ENTER key).

Three levels within each class are defined-- A, B, and C. Level A is composed of those functions within the class that are well

defined and required for a computer terminal to reside in that class. Level B is composed of those functions within the class that are well defined and not required for a computer terminal to reside in that class (advanced functions). Level C is composed of functions that are not well defined and not required for a computer terminal to reside in that class. Level C is meant for special functions that are standardized within a particular installation or organization.

If a computer terminal is in a particular class and level it must support every function defined in that class and level. For those terminals that do not directly support all of the functions defined, there must be some hardware or software simulation of those unsupported functions. If a computer terminal cannot be made to support all of the functions in a class then that terminal cannot be a member of that class and/or level.

The Virtual Terminal Layered Structure

Figure 1 presents a four layer approach to the terminal virtualization. The four layers are the user layer, the simulation layer, the translator/driver layer, and the physical terminal.

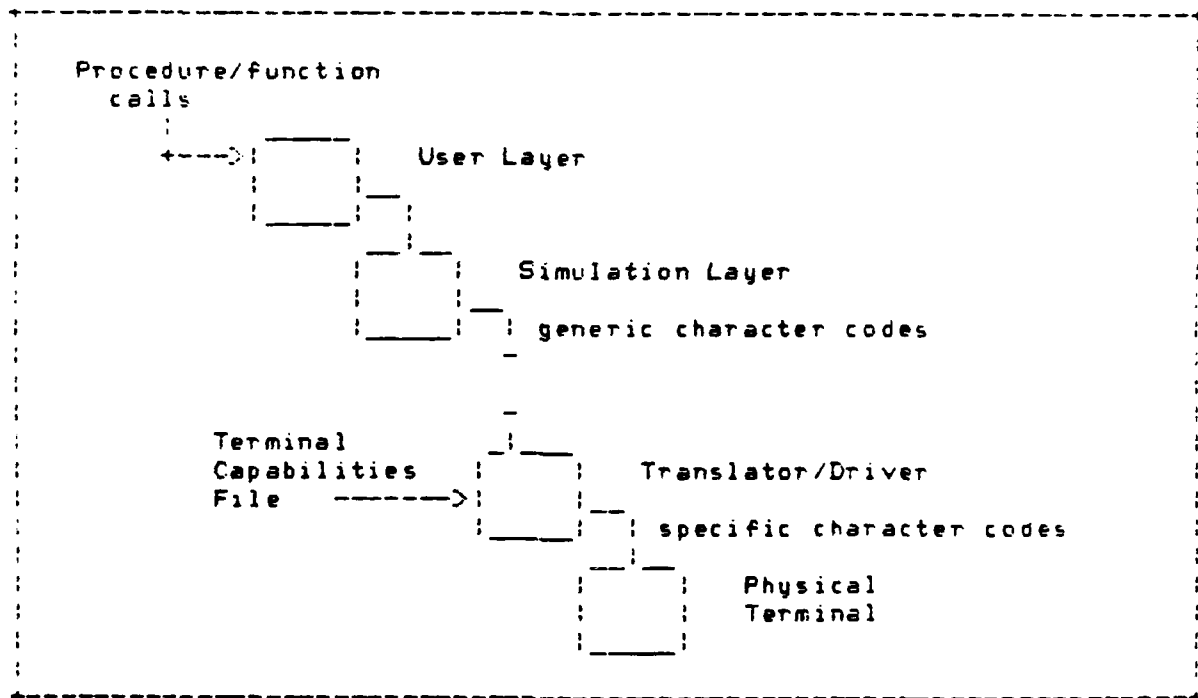


Figure 1 A Four Layer Approach to Terminal Virtualization

The user layer contains the interface that the tool writer sees. This includes all the functions, procedures, abstract types, and exceptions. At this level of the model the functionality of the complex computer terminal is visible.

The simulation layer supplies the software simulation to create those functions that the physical terminal does not support out of those functions that the physical terminal does support. The simulation layer is written in a high level language (such as ADA or PASCAL) to support changes and additions as required.

The translator/driver layer provides the mapping from device independent generic character codes into device specific character codes. This layer incorporates a variation of the UNIX terminal capabilities database which is used to define the mapping.

The physical terminal layer contains the actual physical terminal. It should be noted that only the translator/driver layer has any knowledge of the exact type of terminal that exists. The simulation layer only knows that specific functions are not available and must be simulated using other generic functions. The user layer only knows that the terminal is of a particular class and level.

The generic character codes that are produced out of the simulation layer conform to ANSI standard X3.64 [ANS79]. As computer terminal manufacturers begin to conform to the standard, the translator/driver and the simulation layer will have less and less to do. And, since the ANSI standard has upward compatibility built into it, the entire four layer approach has the same degree of upward compatibility.

Terminal Standardization Efforts

There are two important terminal standardization efforts of interest: the draft standard ISO/DIS 6429.2-1982 [ISO82]; and, the accepted standard ANSI X3.64-1979 [ANS79]. The ISO standard was first proposed as a draft in 1975. It was developed as a synthesis of ANSI X3.64 and ECMA-48 "Additional Controls for Character-Imaging I/O Devices." The ISO standard is a superset of the ANSI standard including additional standardization in the areas of graphic rendition, modes, typographic size selection, and modal interactions. Related standards include [ANS73] [ANS74] [ANS77].

ANSI X3.64

It is natural to use the functionality defined in the accepted standard as the complex model of a character-imaging computer terminal. Acceptance is guaranteed by those that accept the standard and wish an intermediate-level virtualization of computer terminals.

ANSI X3 64 "defines a set of encoded control functions to facilitate data interchange with two-dimensional character-imaging input-output devices" [ANS79]. These control functions may be used in either a 7-bit or 8-bit environment following the code structure defined in [ANS74]. The purpose of X3.64 is to provide a set of control functions to accomodate the foreseeable needs in a variety of information interchange applications: interactive terminals of the cathode ray tube type, interactive terminals of the printer type, line printers, microfilm printers, software usage, form filling, composition imaging, word processing, input-output devices with auxiliary devices, and buffered and non-buffered devices. In the creation of a virtual terminal we are interested in only the interactive terminals and form filling terminals. Perhaps this is too limiting, however, it does produce a nice symmetry and limits the scope a great deal. And, since the virtual terminal does conform completely

with the standard, inclusion of other control functions is easily accomplished.

The Supported Functions

This section presents the functions that the tool writer can use and that form the intermediate-level computer terminal model. Table 1 presents those operations that are well defined and required for a terminal to be classified a class 0 terminal. Table 2 presents additional operations for class 0 terminals, well defined and not required. Table 3 presents those operations that are well defined and required for a terminal to to be classified a class 1 terminal. This class supports most of the terminals on the market today. Table 4 presents additional operations that are well defined and not required, supporting class 1 terminals. Table 5 presents well defined and required operations for a terminal to be classified a class 2 terminal. The semantics of each of these operations is defined somewhat in the ANSI and ISO standards. It is beyond the scope of this paper to give a complete semantic meaning of each of them

There are no additional operations identified for the class 2 terminal. This will probably change as more data is gathered. Also, note that there are no entries for level c in any class. This is intentional, as this level is reserved for installation extensions.

Certainly other classes are possible, they must simply be identified.

Table 1 Class 0a - Scroll Mode Support

<pre>read_line write_line update open close set reset keyboard_action_mode control_representation_mode</pre>
--

Table 2 Class 0b - Additional Functions for Scroll Terminals

<pre>read_character write_character select_graphic_rendition cursor_horizontal_absolute cursor_horizontal_tab cursor_tab_control</pre>
--

Table 3 Class 1a - Page Terminal Support

select_graphic_rendition	select_editing_extent
cursor_horizontal_absolute	edit_in_display
cursor_next_line	edit_in_line
cursor_backward	passthrough_as_is
cursor_down	redraw_display
cursor_forward	
cursor_position	
cursor_up	
delete_character	
delete_line	
erase_character	
erase_in_display	
erase_in_line	
insert_line	
insert_character	
read_character	
read_line	
read_string	
read_display	
write_character	
write_line	
write_string	
write_display	
update	
open	
close	
set	
reset	
keyboard_action_mode	
control_representation_mode	
insertion_replacement_mode	
status_reporting_transfer_mode	
erasure_mode	
vertical_editing_mode	
horizontal_editing_mode	
editing_boundary_mode	
send_receive_mode	
dynamic_update_mode	
line_feed_new_line_mode	
reset_to_initial_state	
get_device_characteristics	
please_report_status	
please_report_current_position	

Table 4 Class 1b - Additional Functions for Page Terminal Support

```
cursor_backtab
cursor_horizontal_tab
cursor_tab_control
erase_in_area
define_qualified_area
    accept_all_input
    accept_no_input_and_do_not_transmit
    accept_graphics
    accept_numerics
    accept_alphabets
    right_justify_in_area
    zero_fill_in_area
    horizontal_tab_stop_at_start_of_area
    accept_no_input_but_select_for_transmission
    space_fill_in_area
read_area
write_area
set
reset
    guarded_area_transfer_mode
    multiple_area_transfer_mode
    transfer_termination_mode
    selected_area_transfer_mode
    editing_boundary_mode
select_editing_extent
edit_in_qualified_area
```

Table 5 Class 2a - Form Terminal Support

```

erase_in_display
erase_in_area
define_qualified_area
    accept_all_input
    accept_no_input_and_do_not_transmit
    accept_graphics
    accept_numerics
    accept_alphabets
    right_justify_in_area
    zero_fill_in_area
    horizontal_tab_stop_at_start_of_area
    accept_no_input_but_select_for_transmission
    space_fill_in_area
read_area
write_area
redraw_display
update
open
close
set
reset
    guarded_area_transfer_mode
    keyboard_action_mode
    status_reporting_transfer_mode
    erasure_mode
    multiple_area_transfer_mode
    transfer_termination_mode
    selected_area_transfer_mode
    dynamic_update_mode
reset_to_initial_state
get_device_characteristics
please_report_status
please_report_current_position
passthrough_as_is

```

Future Directions

Testing out the model presented with real application programmers and tool writers is the direction in which this model will proceed. This will hopefully answer the questions concerning the model. Is it complete enough? Is it too robust? Where are the deficiencies? It will then be necessary to adjust the operations within the classes to more accurately reflect terminals' capabilities.

In the long term, attempts will be made to define new classes to cover terminals that are just beginning to emerge on the market. These terminals begin to approach the functionality of displays found on such workstations as the Apple LISA, Xerox Star, and Smalltalk. Also, support will probably need to be provided for the most simple terminal-like device. An example is an applications in which a device that is not a computer terminal is connected into the physical terminal layer. This could be a hardware debug device or a networking device that needs a completely different model than that presented here. There will be a need to incorporate more operations defined in the ANSI standard as the terminal manufacturers begin incorporating them into their terminals. Along the same lines, consideration should be given to incorporating some of the ISO standard into the

model. Consideration should also be given to the new proposed teletext and videotex standards.

Conclusions

Character-imaging computer terminals are complex devices that need to be treated as such. This paper presents a virtualization of these types of devices to enhance transportability of programs that perform computer/terminal interaction, to provide a common interface for terminals produced by a wide variety of manufacturers, to provide the tool developer with an extensive set of interactive terminal control functions, and to provide the virtualization at a level that supports many different models of computer/terminal interaction.

References

- [ANS73] American National Standards Institute, "American National Standard Graphic Representation of the Control Characters of American National Standard Code for Information Interchange (ANSI Standard X3.32-1973)," July 1973.
- [ANS74] American National Standards Institute, "American National Standard Code Extension Techniques for Use with the 7-Bit Coded Character Set of American National Standard Code for Information Interchange (ANSI Standard X3.41-1974)," May 1974.
- [ANS77] American National Standards Institute, American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)," June 1977.
- [ANS79] American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)," July 1979.
- [BOS83] Bos, J., "Whither Device Independence in Interactive Graphics?," International Journal of

Man-Machine Studies, No. 18, 1983, pp 89-99.

[COR79] "CORE-Status Report of the Graphical Standard Planning Committee," Computer Graphics (Siggraph-ACM), 1979.

[COX83] Cox, F., "KAPSE Support for Program/Terminal Interaction," Working Paper for KITIA/Working Group 1, Feb 1983.

[DAV79] Davies, D., et al., "Computer Networks and their Protocols," John Wiley and Sons, NY, 1979.

[GKS82] Draft International Standard ISO/DIS, Information Processing, ISO/TC97/SC5/WG2 N117 X3H3/82-10, "Graphical Kernel System (GKS), Functional Description," 1982, pp 55, 56.

[GRE80] Greninger, L and Roberts, R, "Considerations for a Logical Virtual Terminal Interface," Proceedings, Distributed Computing, CompCon, Fall 1980, pp 33-40.

[ISO82] International Standards Organization, Standard number: ISO DP 6429, "Additional Control Functions for Character Imaging Devices (Draft)," Not approved, April 1982

- [JOY81] Joy, W. and Horton, M., "TERMCAP," UNIX Programmer's Manual, Seventh Edition, Berkeley release 4.1, June 1981.
- [MAG79] Magnee, F., et.al., "A Survey of Terminal Protocols," Computer Networks 3, 1979, pp 299-314
- [SCH78] Schicker, P. and Duenki, A., "The virtual Terminal Definition," Computer Networks 2, 1978, pp 429-441
- [TAJ79] Tajima, T. and Katsuyama, Y., "Layered and Parametric Approach to Terminal Virtualization," Conference Record, International Conference on Communications, Volume 2, Boston, MA, June 10-14, 1979, pp 22.6.1-22.6.6.

PROGRAM INVOCATION AND CONTROL KITIA INTERIM TECHNICAL NOTE

Anthony Gargaro
Computer Sciences Corporation

This interim technical note refines concepts previously presented in a paper [1] on the program invocation and control interface of the Kernel Ada[*] Programming Support Environment (KAPSE), and reflects ongoing working group participation in support of the following KITIA charter activities [2]:

- further definition of Stoneman [24]
- study of Interoperability and Transportability for Standard KAPSE Interface Specifications
- approaches to KAPSE design and implementation.

In particular, this note elaborates upon some preliminary observations on distributed processing and security with respect to the program invocation and control interface that will enhance tool transportability. The note also documents the author's response to several KITIA communications and working papers that have referenced APSE distribution and security, and the debugging interface.

Earlier drafts of this note were reviewed by members of WG.1 and their comments have contributed to this revised version. A further revision to this note is anticipated; it will respond to comments not currently addressed, and present additional refinements.

[*] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

I Introduction

Recently, there has been an increasing appreciation of the economic, performance, and reliability advantages of interconnecting multiple computers into a network of shared processing resources. The integration of microprocessors for personalized local environments into such networks has accelerated and emphasized the need for shared processing resources. This has necessitated renewed interest in improved system security for shared environments, in the commercial, industrial, and military communities. As a consequence, it has become apparent that the KAPSE Standard Interface Set [25] must provide sufficient semantic detail to ensure the transportability of tools and the interoperability of data, especially if APSEs are to achieve some measure of adaptation to network and secure operations. Through the Interoperability and Transportability (IT) of APSEs data and tools, the potential benefits of secure network configurations can be attained.

Fundamental to supporting such operations is a formulation of the more elaborate tool execution contexts. These contexts must now encompass the additional requirements for distributed and secure program execution. Each of these requirements exact increased demands for tool transportability. For example, within a multilevel secure APSE, a tool must execute concurrently for differently classified users, while within a network of APSEs a tool's execution may migrate among processing nodes in order to optimize resource utilization. This latter requirement adds a dimension to tool transportability that may be characterized as dynamic rather than static. Therefore, KAPSE interfaces would include specification detail that is essential for dynamically sharing a tool's execution among different machines. For conventional status transportability a tool's execution environment is determined when it is called through the program invocation and control interface, and is not permitted to change through migration. For dynamic transportability, tool reconfiguration may be assisted at execution time through some form of dynamic linking [27] in order to adapt the tool to different environment execution contexts.

II Program Invocation and Control

The program invocation and control interface defined in the Interoperability and Transportability Requirements for Standard KAPSE Interface Specifications [21] (IT Requirements) presents an abstraction of user functionality derived from the informal requirements of Stoneman [3], and their supporting interfaces documented in the ALS [4] and AIE [5] B5 specifications. A perspective of this interface was presented in an original point paper [6] in the framework of code execution services for the APSE. In this canonical framework, program invocation and control is distinguished by the granularity, or scope, of the thread of control that is managed by the APSE-KAPSE interface. The scope of the thread of control for the program invocation and control interface is normally defined to be an Ada main program. According to the Ada Definition [18], a main program acts as if called by some environment task, and is a subprogram that has been compiled as a library unit. Applying this informal description, an APSE tool is a main program that qualifies as an environment task, if it uses the program invocation and control interface to call a main program. However, because an environment task is not defined, several interpretations of this interface have been identified for further study in order to motivate tool transportability. In addition, other legitimate threads of control that comprise the code execution services are derived from the semantics of the Ada task model, and the services to support the monitored code execution contexts required by the debug tool [19].

This perception of code execution services has become useful with the elaboration of distributed processing, security, and extensibility considerations on the KAPSE interfaces by the IT Requirements. These considerations were not specifically included in the Stoneman definition, and therefore, were not emphasized in the KAPSE interfaces of the current designs for an APSE.

III Capacity Transparent Interfaces

In the referenced point paper, the notion of Capacity Transparency was introduced as a means to improve tool transportability. An elegant exposition of capacity transparent interfaces is presented in the Ada definition of numeric real types. Arithmetic operations are guaranteed consistent accuracy (semantics) across different compilers by defining all real operations (interfaces) in terms of model numbers. The different compilers may choose the most efficient implementation defined standard accuracies, from which the user specified accuracies are derived using the rules of model arithmetic. The accuracy of intermediate results is transparent to the arithmetic interface since it depends on the capacity of the target arithmetic hardware. Similarly, Capacity Transparency should be applied to the code execution services to accomodate both static and dynamic transportability, and to retain the flexibility deemed essential for potential interface extensions necessary for developing more portable run-time support for Ada tasks [7,12]. As a result, additional processing capacity compatible with the interface semantics may be used. The specification of the interfaces may specifically limit execution through an appropriate processing constraint. For example, the permissive semantics of invoking a program must be refined to state whether or not the execution of the called program occurs on the same processing node as the calling program, since its execution may depend upon data accessible at the calling node (e.g., efficient implementation of a program pipe usually requires that the pipe couples programs executing on the same node). The Process Management packages of the Standard Interface Set have included a location parameter to define an implementation dependent constraint when invoking a program. In addition, Capacity Transparency is compatible with the promising approach suggested for designing and formalizing interface semantics using the denotational method [16, 17].

IV Program Distribution and Security

A key concern in specifying a durable interface for program invocation and control is an understanding of the requirements for distributed program execution, and the security of program execution in the target systems that may be associated with the APSE. The requirements for targets are introduced because the functionality to achieve distribution and security is a fundamental need in tactical environments, and is outside the province of the conventional Ada run-time support library [13]. This provides adequate justification for specifying a KAPSE-like analogue on the target computer(s) termed the Target Ada Support Kernel (TASK). This suggests that standard KAPSE interfaces be developed that are adaptable as interfaces to the TASK, thereby not only facilitating tool transportability among APSEs, but also their (restricted) use on target systems. The increasing emphasis for enhanced target debugging capabilities in future multilingual environments [26], indicates a need for tool transportability to target systems.

An important contribution of consistent interfaces for the host and targets derives from those code execution services that allow the run-time support for Ada task execution to be more readily transportable. The critical performance requirements for code execution in the target are delegated to the TASK, independent of the high level Ada task semantics; a significant advantage when the target must execute programs from different language environments [26]. A taxonomy of KAPSE interfaces, that is realized through generic units, packages, and library facilities, is envisaged to provide categories of interface functionality. These categories do not conform to a layered KAPSE [20] approach, which is primarily an implementation rather than specification strategy, since only the outermost layer is included in the APSE-KAPSE interface. The extensive use of Ada to formulate functional categories is commensurate with the expected trend that encourages the application of programming language principles in the design of software systems [8].

The adaptation of KAPSE interfaces for totally distributed and secure APSEs extends beyond proven software technology, and understanding the implications of such an adaptation are not readily quantified, especially for dynamic transportability issues. Therefore, as an initial step, useful

principles extracted from existing and experimental designs can be factored into preliminary interface requirements and KAPSE design by addressing distribution and security issues as they relate to code execution services. Specific requirements for the different levels of computer security have been summarized [14] to provide sufficient guidance in developing the rudimentaries for the execution of tools in a secure APSE. KAPSE interface specifications for program execution should address the classification of tools, the creation of protected execution contexts, intertool communication in a multilevel APSE, access to global (system) data, and the verification of the KAPSE implementing the interfaces.

To demonstrate a potential security flaw in the semantics of existing program invocation and control interfaces, the CALL tree maintained by the ALS KAPSE can be cited as an example. The semantics of program call and program control would permit the leakage of information (i.e., the confinement problem) through the CALL tree, even if the interfaces were refurbished to support multilevel program execution to monitor calls and parameter passing. In this instance, the CALL tree becomes an indirect storage channel through which information can be transmitted by a classified program through the interface semantics (e.g., adding and deleting nodes), and then subsequently addressed by an unclassified program.

One of the challenges presented by Ada is that of supporting the execution of a multilevel secure program, i.e., a program consisting of Ada tasks that perform operations at different classification levels. For tactical applications secure task execution may be achieved by an approach that relies on suitable Ada security pragmas, and the use of the code execution services of the TASK by the run-time support library to establish secure task execution contexts [15].

V Distributed APSE

Before the security policies and mechanisms relevant to an APSE are investigated further, it is necessary to delineate the degree to which an APSE can execute on a distributed configuration. A recent study [9] has presented substantial arguments for secure systems to be conceived as distributed systems, where security is achieved partly by the physical separation of the individual components (tools), and partly by the trusted functions performed by some of the components. As a result, the various security requirements of APSE components are recognized and enforced by the individual component. The APSE becomes a network of cooperating, distributed, and independent programs, where physical separation of each program provides an alternative to the conventional security kernel. The specification and verification of the properties of the trusted functions are APSE design issues, while the functionality to guarantee the distribution and separation of the components are seen as KAPSE design considerations for program invocation and control. This approach is to be elaborated upon in subsequent technical notes, but in the interim it will be used to influence and justify decisions regarding distributed processing capabilities for the APSE that are motivated from both a logical and physical perspective.

The Stoneman report does not directly address the requirements for a distributed APSE. A general guideline proposed is that the APSE should be designed to exploit the underlying hardware of the host system. Presumably, if the hardware comprises a network of interconnected machines, the KAPSE is expected to provide sufficient functionality to utilize the network efficiently. For example, a user may logon to a local node of the network, and yet have session processing performed by any node in the network that satisfies the processing resource requirements. Such a capability would be characteristic of a KAPSE that provided capacity transparent interfaces that guarantee consistent tool execution on any node selected. The attainment of this level of distribution is not anticipated in either of the current APSE implementations. Each design has recognized that distributed processing requirements must not be obviated by an overly restrictive interface. This is particularly evident in the KAPSE for the AIE that specifies both program invocation and program

communication services in the 'KAPSE/Tool' interface. A criticism in an earlier note of this design regarding apparent host dependencies in the interprogram communication services is resolved by the revised design [5]. 'KAPSE/Host' communication services are specified that enable a tool to exchange information with the KAPSE across a protected boundary. These services are distinct from the tool's interprogram communication services, thereby preserving the transportability of the tool interface. In addition, a 'KAPSE/KAPSE' interface is proposed to facilitate distributed APSE support. However, totally distributed code execution services are precluded by the suggested semantics of the 'KAPSE/Host' interface, and the integration of task execution within the run-time system included in each tool. The interface relies on Ada tasking to effect a communication protocol between a tool and the KAPSE program. It uses a communication task in each tool and a service task within the KAPSE, thereby excluding the distributed execution of tasks through the KAPSE and compromising the transportability of the run-time system.

As a consequence, it is necessary to stipulate formal guidelines for studying the performance of an APSE on a physically distributed host system. For convenience, the term 'distributed processing' is used to denote any hardware environment that supports either logical or physical separation of processing. Justification for this definition has been suggested earlier, and will be argued subsequently in terms of functional isolation for system security.

The need to consider one aspect of the physical separation of processing was identified in the Stoneman report as a result of down-line testing requirements [10]. This style of testing envisaged a target resident program under the supervision of a host resident debug tool, thereby distributing the APSE functionality between the host and target machines. Thus, the host and target form a limited, but legitimate, distributed configuration, especially when the target is another host that is used to provide the required resources necessary for testing the application program.

The outcome of this observation has an impact on the program monitoring interface that must now be perceived as a variant of the program invocation and control interface. The debug tool communicates and controls the executing

application program through a debug implant that has been linked with application object code. As a result, this variant motivates the need for a consistent extension to the Program invocation and control interface category to include semantics for distributed processing. In addition, it establishes the rationale for the APSE to be distributed on different machines since multiple users may wish to test programs on distinct or heterogeneous targets. The case of rehosting the APSE is an excellent paradigm for down-line testing. The user employs the debug tool executing on the host to test a tool rehosted on the target (the new APSE host). If the rehosted tool is the debug tool, it may in turn be testing a program executing on the host which is then considered a target. In this instance, the functionality of the APSE is not only distributed, but is configured in the role of both a host and a target environment.

The expected development and proliferation of personalized local workstations to improve software productivity favors the concept of an APSE that is distributed among an APSE Hub and a local network of personal computers. Only the simpler tools of the APSE that have restricted data demands would execute on the Hub after having been invoked from a node. It is interesting to observe that in this configuration if the node was a compiler target, then compiled programs could be debugged locally. This increases the KAPSE-TASK intersection of functionality, and substantiates the artificiality of the severe distinction that has classically separated a host from a target. Consequently, the functional intersection facilitates the run-time support library to define a homogeneity level among hosts and targets, which on one side, allows application testing, and on the other side, can provide partial environment services on the targets [27].

VI Distribution Models for the APSE

In the context of the code execution services elaborated in the referenced point paper, distribution of processing may be considered at two levels of user visible functionality, interprogram and intraprogram. Interprogram functionality is representative of the program invocation and control (including the program monitoring variant) services that are essential

to support the Command Language Processor, and the requirements for tool synergism. This is clearly demonstrated by the minimal facility through which the Command Language Processor enables a nonstandard command language to be made available in the APSE, and the execution of command language procedures by the Configuration Management System [22]. Intraprogram functionality is representative of the services required to support Ada tasking that are profitably implemented outside of the run-time support library. This suggests that different models of logical and physical distribution should be considered in specifying the code execution services interface. The different models are used to define classes of code execution services that a conforming KAPSE implementation may provide. The transportability of a tool would depend upon the extent to which it uses the semantic differences these classes introduce upon an interface. A common example might be the invocation of a tool (e.g., the Editor) that is to read the standard input file of the calling tool (viz., the Command Language Processor). The semantics of the interface used must guarantee that the called tool executes on the same processing node as the calling tool. Omission of such explicit semantic detail can compromise the transportability of the calling tool. The calling tool safeguards its transportability through the necessary processing constraints. These constraints would be materialized through the semantics of the interface class. Classes for a particular interface might be developed through customized packages encapsulating the subprogram specification. Figure VI-1 outlines a possible specification strategy. All code execution services are encapsulated in a package. The specification of this package includes generic package specifications that are available for the proposed levels of program management functionality. An instantiation of these generic packages for the class of program distribution required by the tool may then be included in the standard environment of the tool through a context clause. The objective of this approach is to reduce the changes to a tool's interface to the KAPSE, through tool recompilation, should a different class of program distribution become necessary. In the example, a tool is shown to make an explicit call to interprogram management to invoke a main program. An implicit call to activate a task is shown as a result of executing an allocator for a task access type or because of the elaboration of a task object. This implicit call is serviced by the run-time support library that has been implemented to use the appropriate instantiation of the intraprogram management package. In this instance, the

run-time support library is treated as another transportable tool, except that it is not invoked as a program. The manner in which the run-time support subprogram bodies are integrated with a program becomes an implementation strategy, while maintaining consistency between the class of distribution services required by the tool and the run-time support library is an important, and as yet unresolved, configuration control issue. The formulation of classes is similar to the use of the representation specifications in the Ada language. The classes provide a means through which the interface specifications can be selected to yield semantic detail compatible with the host system resources. If used inappropriately, the transportability of a tool is compromised through the violation of the capacity transparent interface property.


```

package KAPSE_CODE_EXECUTION_SERVICES is
-- Code execution services for all APSE distribution models.
generic
    package INTER_PROGRAM_MGMT is
        -- For use by all tools, includes Program Invocation
        -- and Control Interface.
        procedure Call_Program ( ... );
    end INTER_PROGRAM_MGMT;
generic
    package INTRA_PROGRAM_MGMT is
        -- For use by the Run-Time Support Library.
        procedure Call_Task ( ... );
    end INTRA_PROGRAM_MGMT;
end KAPSE_CODE_EXECUTION_SERVICES;

with KAPSE_CODE_EXECUTION_SERVICES;
package CLASS_N_PROGRAM_MGMT is new
    KAPSE_CODE_EXECUTION_SERVICES.INTER_PROGRAM_MGMT ( ... );

with KAPSE_CODE_EXECUTION_SERVICES;
package CLASS_N_TASK_MGMT is new
    KAPSE_CODE_EXECUTION_SERVICES.INTRA_PROGRAM_MGMT ( ... );

with CLASS_N_PROGRAM_MGMT; use CLASS_N_PROGRAM_MGMT;
with CLASS_N_TASK_MGMT; use CLASS_N_TASK_MGMT;
package body RUN_TIME_SUPPORT_LIBRARY is
    procedure Activate_Task ( ... ) is
    begin
        Call_Program ( ... );      -- Explicit call.
    end Activate_Task;
end RUN_TIME_SUPPORT_LIBRARY ;

```

KAPSE SPECIFICATION STRATEGY
Figure VI-1

Class-0 formulates a basis for logical distribution where the code execution services are provided on a single machine (uniprocessor) with no facility for physical down-line testing. The machine is multiplexed among separate threads of control for interleaved execution of programs and tasks. This achieves the minimum properties required by the APSE. From this basis additional classes (1..6) are derived to introduce increased physical processing distribution for interprogram and intraprogram functionality.

Class-1 provides for physical down-line testing as described previously. The code execution services of the KAPSE are upgraded to facilitate the initiation and control of an application program on the target machine. The debug implant in the application program uses a KAPSE compatible interface that must be supported on the target machine. Through the code execution services interface, the program is loaded, initiated, and controlled via the implant. The functionality of this interface relies upon the concept of target software (i.e., the TASK) that is separate from the run-time support included in with the application program. When necessary, the TASK will allow the target machine to execute multiple programs in those instances where the target machine is a critical resource, and must be shared among several users of the APSE debug tool.

Class-2 is derived to support separate threads of control on a multiprocessor configuration. Both interprogram and intraprogram functionality are revised to exploit the opportunity for improved performance through parallel execution of programs and tasks. It is expected that this upgrade would not change the KAPSE interface specification, but would influence its design and implementation. For example, task dispatching would not be accomplished by the run-time support system. A suggested reorganization of the ALS RSL/KAPSE design for efficient multiprocessor support was outline in the reference point paper.

Class-3 is derived to support the physical distribution of APSE tools. Interprogram code execution services are enhanced to facilitate communication among APSE tools executing in a network of homogenous nodes (machines). These services require a more elaborate interface since identification of program executions (processes) is essential if interprogram communication is to include more than the parameter exchanges at program invocation and termination.

Process identification has been perceived as a manifestation of the overall binding problem [23] that must now encompass node identification in a manner consistent with maintaining capacity transparent interfaces. Each node (in its role as a host), is required to execute a family member of the same implementation of the KAPSE, and a program execution environment is constrained to a single node. For example, the Command Language Process might have multiple executions on every node, whereas the compiler may be restricted to nodes with sufficient resources; e.g., the Ada Library may be dedicated to a specific node. In this instance, invocation of the compiler would attempt initiation on the appropriate node. Extension to the code execution services would be accompanied by significant enhancements to the KAPSE-Host interface to support distributed processing supervision, network topology management, packet handling, and protocol services.

Class-4 is derived from Class-3 to support the physical distribution of Ada tasks on machines hosting the same KAPSE implementation. Intraprogram functionality is upgraded to activate and control the execution of Ada tasks on separate machines. Significant revisions to the KAPSE and run-time support library must be implemented to compensate for the fragmentation of a shared program environment. Popular implementation strategies based upon procedure calls [11] would become obsolete, and many severe problems associated with its feasibility are anticipated. The difficulties of distributed tasks warrant elaboration in a future technical note.

Class-5 and Class-6 dominate Class-3 and Class-4 respectively, and are derived to facilitate APSE tools executing in a network of heterogeneous nodes. Reduced capability environments executing on personal computers connected to an APSE Hub would be characteristic of Class-5 distribution. Justification for Class-6 can be argued from the documented requirements for the execution of Ada tasks on different interconnected target computers [26], coupled with the need for the KAPSE and TASK interfaces to converge functionally.

```

package TASK_CODE_EXECUTION_SERVICES is
--Code execution services for Class-1 distribution model.
  package INTER_PROGRAM_MGMT is
    package DEBUG_SERVICES is
      procedure Send_Host ( ... );
    end DEBUG_SERVICES;
  end INTER_PROGRAM_MGMT;
. . .

  package INTRA_PROGRAM_MGMT is
    procedure Call_Task ( ... );
  end INTRA_PROGRAM_MGMT;
end TASK_CODE_EXECUTION_SERVICES;

with TASK_CODE_EXECUTION_SERVICES;
use TASK_CODE_EXECUTION_SERVICES;
procedure Tactical_Program is
  task Some_Task is
    pragma DEBUG ( On= Downline );

  end Some_Task;
  task body Some_Task is
    use INTER_PROGRAM_MGMT;
  begin
    Send_Host ( ... ); -- Implicit call from implant.
  end Some_Task;
begin
  Activate_Task ( Some_Task, ... );
end Tactical_Program'

```

TASK SPECIFICATION STRATEGY
Figure VI-2

Although different KAPSE implementations would exist in a network, each KAPSE would be required to supply a proper subset APSE-KAPSE interface. For example, one KAPSE, hosted on a multiprocessor, might provide in addition to Class-5, a Class-2 interface, whereas another KAPSE hosted on a personal computer would only provide the stipulated Class-5. Class-1 can be viewed as a degenerate case of Class-5, where the TASK supports a subset of the Class-5 interface. Other classes would include distribution models that provide support for dynamic transportability.

VII Conclusions

This interim technical note has proposed several areas for continuing study that appear useful in formalizing requirements for the program invocation and control interface to promote tool transportability in both a distributed and secure environment. The reader should understand that one of the primary purposes of the note is to stimulate views that may be radically different from those expressed in this note. Furthermore, it is appreciated that some of these views are controversial, and currently lack the technical rigor necessary to stipulate either a KAPSE interface or a specific set of formal IT requirements.

In summary the following points are emphasized:

- The program invocation and control interface requirements should be sufficiently well specified in order to avoid unacceptable utilization of the host system. The corresponding interface must evolve into a more comprehensive set of services, and should exhibit a degree of capacity transparency.
- The run-time system should be transportable to the same extent as a tool. It should also be adaptable for use on a target machine, through the functional intersection of the KAPSE and TASK interfaces.

- The distribution and security of an APSE is a major requirement of future program invocation and control interfaces. It is proposed that security may be achieved partly through an interface that supports the logical and physical distribution of tools.
- There are potential advantages in developing an interface that is compatible with its use in a target environment.
- The formulation of APSE distribution models for which IT is deemed appropriate is recommended. These should be included in the IT Requirements.
- A KAPSE specification strategy should be developed that is consistent with the extensive specification facilities offered by the Ada language.

Bibliography

- [1] Program Invocation and Control; KIT: Public Report Vol. 2; Technical Document 552; 1982-10-28.
- [2] Preliminary KITIA Charter; 1982-04-30.
- [3] Requirements for Ada Programming Support Environment; Stoneman; 1980-12.
- [4] Ada Language System; KAPSE B5 Specification; Softech, Inc.; Report CR-CP-0059-B83; 1982-02-28.
- [5] Computer Program Development Specification for Ada Integrated Environment; KAPSE/Database Type 65; Intermetrics, Inc.; Report IR-678-2, B5-AIE(1).KAPSE(1); 1982-11-12.
- [6] Point Paper F; KIT: Public Report Vol. 1; Technical Document 509; 1982-04-01.
- [7] Ada Integrated Environment: Computer Program Development Specification, Part 1; Computer Sciences Corporation and Software Engineering Associates Inc.; 1981-03-15.
- [8] SIGPLAN'83 Call for Papers - Programming Languages Issues in Software Systems.
- [9] Rushby, J. M.; the Design and Verification of Secure Systems; Proceedings 8th. ACM Symposium on Operating System Principles; 1981-12.
- [10] Fairley, R. E.; Ada Debugging and Support Environmental; SIGPLAN Ada Symposium; 1980-12.
- [11] Habermann, A. N./Nassi, I. R.; Efficient Implementation of Ada tasks; CMU-CS-80-103; 1980-01.
- [12] U.K. Ada Study; Final Technical Report Vol. 4; Department of Industry; 1981-06.
- [13] Ada Language System; VAX/VMS Runtime Support Library; SofTech, Inc.; Report CR-CP-0059-820; 1982-02-24.
- [14] Landwehr, C. E.; Formal Models for Computer Security; Computing Surveys, Vol. 13, No. 3: 1981-09.
- [15] Proprietary Comany Material; Computer Sciences Corp.
- [16] Point Paper E; KIT: Public Report Vol. 1; Technical Document 509; 1982-04-01.

- [17] Ashcroft, E., Wadge, W., ; Rx for Semantics;
ACM Trans. Program. Lan. Syst. 4/1; 1982-04.
- [18] Reference Manual for the Ada Programming Language;
ANSI/MIL-STD 1815A; 1983-02-17.
- [19] WG.1 Meeting Minutes (Blacksburg); 1982-08-10/11.
- [20] Wrege, D.; Layered KAPSE; KITIA Meeting (Waltham);
1982-06-20.
- [21] Requirements for Interoperability and Transportability and
Design Criteria for Standard Interface Sets; 1983-02-18.
- [22] Configuration Management System; Interim Report on
Interface Analysis; Computer Sciences Corp.; 1982-08-27.
- [23] Kramer, J.; Binding Category; Arpanet Message; 1982-08-09.
- [24] Milton, D.; Stonemen II; Draft 1: 1983-01.
- [25] Ada Package Specifications for the Standard Interface Set;
Draft 1, Version 1: 1983-02-01.
- [26] Ada Language System/Navy; System Specification; 1982-10-15.
- [27] Iverardi, P. et.al.; A Distributed KAPSE Architecture;
Ada-Europe/AdaTEC Joint Conference on Ada; 1983-03-16/17.

SIS IMPLEMENTATION ISSUES PARAMETER PASSING OVER THE STANDARD INTERFACE

H. Fischer
Litton Data Systems

Abstract: This note discusses physical implementation of the "Standard Interface" also known by the KIT as the SIS. Use of the Ada Language parameter passing mechanism and type conversion mechanism is shown to be a useful mechanism for implementing the calls across the "KAPSE Boundary". It is shown that applications programs, written in different environments with different parameter storage variables, can utilize Ada Language features to automatically correct the parameter appearance to the routines on the inside of the "KAPSE Boundary", thus avoiding need to tweak source code calling sequences when porting applications systems.

The KAPSE Boundary

Using the standard KAPSE model, the KAPSE is a set of standardized system services usable by tools in a MAPSE and tools and programs in an APSE. The KAPSE services will be defined by the KIT/KITIA, initially as a derivation from the "common intersection" of AIE and ALS, and later by an in depth analysis. KAPSE services include database (file) access and management, program initiation and termination, and input/output services. They also include host-dependent services such as storage management, virtual storage implementation, hardware interrupt handling, and the like.

The "outside" of the KAPSE boundary is the user's domain, which in the program support environment, includes both the user's own programs and the tools to create and maintain the programs (compilers, debuggers, command line interpreters, and editors).

Most (maybe all) of the published papers referring to KAPSE implementation describe the boundary between KAPSE and external world as described (in part) as Ada packages[1],[2]. Actually, it is only the syntactical appearance of the boundary which is easy to describe as Ada packages[2]. The syntax describes what the boundary looks like to coding. It is the syntax which the balance of this paper will address, because that is the part which the implementors will provide, physically, to create the KAPSE "boundary" in code. First, however, it is necessary to note that the semantics of the interface, the specification of the meaning and intent of the KAPSE "calls", are an entirely different issue. One can, of course, describe semantics in English prose. That is how the Ada manual describes Ada features. English prose is most easily understood by nonscientists. On the other hand one can describe the semantics in terms of models,

representations based on the use of symbols and/or a notational system which avoids ambiguities and language difficulties. On the third hand one can describe the semantics in terms of rules, using forms of predicate calculus to create a model of the KAPSE. (A predicate calculus or rules description is generally not readable by the public at large, and thus is separated from a notational form of model. It is often, however, implementable (modelable) on the computer, and can be used to prove correctness.)

Physically Building the Syntax of the Interface

Ada facilitates building syntactical descriptions of interfaces, in the form of package specifications for the "code bodies" which implement the KAPSE services (or which interface to other facilities which perform the KAPSE services). These interface descriptions are packagable separately from the code bodies of the KAPSE routines, which means that source code and compiler-readable forms of these interfaces can be distributed to KAPSE users without distributing corresponding source or reconstructable forms of the KAPSE itself. This is an important factor when dealing with systems whose security would be compromised if users had access to internal construction details of the system.

The interfaces are described by three items: (1) specification of the KAPSE service, (2) specification of the subtypes of the formal parameters, and (3) specification of the types referred to in the subtype and formal parameter descriptions.

KAPSE services are identified by subprogram declaration. The subprogram declaration identifies the name of the KAPSE service as well as the names, modes, and type marks of formal parameters (arguments) of the service. (Default values of parameters may also be specified.) A mode declaration denotes whether a given parameter is supplied to the KAPSE by the user, returned to the user from a KAPSE service, or both. A type mark names the type or subtype which describes the nature of the parameter.

The subtype of a parameter to a KAPSE service identifies the more general type and constraints on the more general type specification. Subtypes identify constraints, such as numeric value ranges and enumeration restrictions (midweek is a restriction on weekday which is a restriction on days). Subtypes may be private, for those items which the KAPSE passes to the user, such as file handles, only for purposes of identification and tracking.

A type describes the most general nature of a set of subtypes which might be more restrictive descriptions of formal parameters.

A user call to a KAPSE service must present parameters required, but they need not be in the implemented order or in the proper format. Ada allows different implementors of different KAPSEs to have their own internal structure. Each KAPSE must provide only interfaces which use formal parameters of the standardized names, and of types which are either (1) explicitly convertible to user forms or (2) private (e.g., not disclosed to the applications user). The type mark for the parameter must have a standard name. A KAPSE service which has a subtype which is not directly usable by the applications program will need to be "overloaded" with a dummy conversion routine, or else the

type mark will need to be explicitly named with the instance of each formal parameter to force explicit conversion. (The way to go is up to the standards definers.) For example, a KAPSE whose internal implementation requires a given parameter to be a character string, but where the standard defines a numeric type, must either overload the subprogram declaration with a "go between" which performs the conversion, or the standard must require users to demand the conversion explicitly by naming the type mark with the parameter in parentheses (explicit conversion in all cases).

The Ada language allows the syntax of package interfaces to be stored in libraries and referenced by "with" statements. The content of a referenced library package would be both statements describing the types and subtype constraints on interface parameters, as well as statements describing the sequence, mode, and types of parameters of each KAPSE "boundary crossing" call.

A user of a KAPSE service need not know the details of internal representation of parameters passed across the boundary; he need only know the names of the parameters and sufficient description of the type of parameter (e.g., character string versus access pointer) so that the parameters he names are type-convertible to conform to the KAPSE implementation requirements. Furthermore, there are many parameters, such as file identifiers, which are both provided by, and used by the KAPSE, which the user has no need to understand. These will most likely be described to the user as "private" types, merely having assignment and equality operators.

Using the Interface

A user of a toolset or application system learns what the KAPSE services can do by reading semantic descriptions (prose, model notation, or logic rules descriptions). The user causes the services to occur within his code by referencing ("withing") the library module containing the particular implementation's type, subtype, and procedure declarations. The user names the services as function or procedure calls to cause the action desired to happen. He identifies parameters by their name (or positionally), and, because the compiler has the particular KAPSE's parameter details available, it can generate code compatible with a particular form of the KAPSE.

Porting Applications to Another KAPSE

When porting from one KAPSE to another, the entire set of applications and tools using KAPSE services will be recompiled, allowing the compiler to use the library module of the new KAPSE's parameter and procedure interface descriptions. Recompile is needed because the user can be expected to refer to the types and subtypes provided for the KAPSE as he manipulates parameters, file identifiers, and the like. The code generated within a tool might thus be different internally, having alternate type conversions, even possible explicit conversions required, depending of the particulars of an example.

References

- [1] "Specifying KAPSE Interface Semantics", R. Freedman. KITIA Group 1.
- [2] "Validation in Ada Programming Support Environments", D. Kafura, J. Lee, T. Lindquist, and T. Probert. page 22 ff.

SIS CATEGORIES

D.E. Wrege
Control Data Corp.

Abstract

This paper proposes that the Standard Interface Set (SIS) be divided into several non-overlapping categories, therefore allowing several classes of Ada Programming Support Environments (APSEs). Further, a set of non-functionality based criteria are proposed to guide the establishment and evolution of the SIS.

Introduction

The requirement for a standard interface at the KAPSE level to provide interoperability and transportability (IT) of tools has been a hotly debated issue. There have been few clear cut criteria or guidelines produced for determining the level of a KAPSE interface, what functions should be included, and whether they should be standardized. These problems must be resolved before many APSEs are implemented and IT can be actually realized.

Definitions

In that there is a profusion of terms referring to KAPSE interface standards (KAPSE, Standard KAPSE, SIS, BASIS, KISS, etc.) the following terms and definitions will be used.

KAPSE: The words KAPSE and KAPSE interface will be used to refer to the interface level of an APSE that encapsulates host dependencies. It is the virtual operating system interface definition upon which it is possible to build all APSE tools in a portable manner (i.e. utilizing only implementation independent Ada language features). The implementation of the KAPSE is, in general, not required to be portable or even necessarily implemented in Ada, although portions may be. Note that the word standard does not appear in this definition.

SIS: The Standard Interface Set is one or more sets of interface definitions and their semantics that are standardized to provide transportability of APSE tools (and interoperability of APSE databases). Note that the word standard is crucial in this definition.

The Portability Non-Issue

If a SIS is adopted and there are multiple APSEs using instances of that

interface, then tools depending only on that interface and utilizing only the machine independent features of Ada will be portable between them. Regardless of the level of the interface, the fact that it is standard ensures portability. Thus the portability issue is solved given the adoption of the SIS approach (and there seems to be no controversy over the establishment of a SIS). The remaining problem is to determine what should be included in the SIS. This determination depends on issues other than portability.

Concerning SIS Inflation

Consider a portable tool running within an APSF. If it has considerable complexity, it should be composed of many separate compilation units. One should not expect that all or even most of these packages represent SIS interfaces. Likely, if the dream of reusable software becomes a reality, some of these packages will also be used by other tools. However, the mere existence of a routine used by many, or even all tools, does not establish any necessity for that routine to be located in the SIS. Indeed, so long as the procedure is portable, the portability of the tool is unaffected by whether the routine is in the SIS or not. Thus, the SIS is NOT defined to be all the stuff that tools use!

Further, just because a procedure is part of a SIS or KAPSE interface does NOT mean that it is somehow memory resident or even implemented in a host dependent way. Indeed it can itself be portable Ada code, dependent only on more primitive host dependent routines. It could even, like many Run-Time-Systems, be linked with the tool as would any other library routine.

Some Guidelines for KAPSE Interfaces

For a particular host there will be some KAPSE boundary that best utilizes the capabilities of that host. By choosing such a boundary carefully the amount of "dirty" code can be minimized while maximizing the efficiency of the resulting implementation for that host. Thus the SIS concept should allow for the possibility of defining a different (optimal) KAPSE boundary for each different host. Clearly, the SIS should be located outside of the locus of optimal KAPSE boundaries of all potential hosts.

Rehostability considerations drive the KAPSE interface toward a single minimum KAPSE boundary. This should be clear since the cost of rehosting a small KAPSE will be less expensive than a larger KAPSE. Security considerations also influence the location of the KAPSE interface. For example, to protect classified database objects, the database access functionality must be behind the KAPSE boundary or the protection mechanism might be circumvented. Therefore security considerations could provide a requirement to drive the KAPSE interface to include certain high-level functionality. Once again, the SIS must encapsulate such KAPSEs.

There are, of course, many additional considerations that must be used in determining the SIS. The above examples are intended to illustrate the nature of such guidelines and to point out that portability is really not the major issue once it is decided that there will be standard interfaces.

The SIS and Its Relationship to the KAPSE

There is the tendency, when considering the STONEMAN concepts of the KAPSE/MAPSE/APSE, to place everything that is portable into the MAPSE (or APSE) and place everything else into the KAPSE. Thus the KAPSE (or SIS) interface encircles all non-portable functionality. Then the erroneous assumption is often made that, for the tools to be portable, this entire interface must be standardized across all APSEs. Indeed, the portion of this interface that portable tools depend on should be standardized, but not necessarily the rest.

The KAPSE will contain interfaces which are not related to interoperability or transportability of tools. For example, standard logon/logoff services are not required to insure tool portability. It is certainly not unreasonable to have a few tools that are host dependent, or some portions of tools with well identified environment dependent parts. After all, 100% transportability may be an important goal, but as is often the case, the last few percent may not be cost effective. As is stated in the draft "KIT Strategy Statement", "It is generally agreed that 100% IT is not likely to be achieved and is not a realistic goal. The real goal ... is to make the sharing of tools and data bases sufficiently practical and cost-effective for sharing to become the normal mode of operation between the various agencies of the DoD as well as the industry which supports them."

Portable tools mean nothing if the APSE is so expensive to rehost, due to the requirement for implementation of a large SIS, that it can be supported on only one machine. In such an extreme case there will be no place to port the tools.

Categories of the SIS

The final argument is for multiple categories of the SIS. There is surely a large collection of tools that depend on some subset of the SIS. For example, most of the current MAPSE toolset needs only fairly simple access to the database, file I/O, and primitive terminal I/O. I suspect that most of the tools in an APSE will fit into this category. Some tools, or collection of tools, will require additional facilities. For example, tools like the APSE Interactive Monitor (AIM) being designed by Texas Instruments require interactive capabilities and process communication capabilities far beyond what most tools need. Should it be required that all conforming implementations of the SIS contain these latter interfaces? Is an all or nothing choice reasonable? It seems that an APSE with all tools except the AIM is certainly a useful environment, although one might prefer one with interactive tools like the AIM.

Consider a strategy where several categories of SIS are defined. A Basic SIS would consist of those interfaces necessary for the Ada Language Tools. An Interactive SIS would contain interfaces that highly interactive tools might need. An Advanced SIS might be defined for some class of advanced tools, and perhaps an Expert SIS would exist for super-advanced artificially intelligent tools. I tend to think of these as separate interface sets, where a union of one or more sets defines the class of an APSE, (i.e. Basic APSE, Interactive APSE, Advanced Interactive APSE, etc.). Note that having such categories of SIS is a familiar concept to current builders of portable programs. A conscious decision is made, with

appropriate tradeoffs, as to what level of dependency a tool should have (e.g. FORTRAN-II, FORTRAN-IV, or FORTRAN-77). The point is that a toolsmith could and should design his tool with careful consideration of which SIS categories are to be required. Also, such a philosophy would lead to a natural mechanism for extensions of the SIS as environments become better understood.

Guidelines for SIS Definitions

Most attempts to define SIS requirements and criteria have revolved around defining functionality. Here it is assumed that interfaces providing certain functionality have been determined. The question is: "Should a particular interface item (or function) be included in the SIS?"

For an interface item to be included in the SIS:

1. The item must not be implementable in terms of existing interfaces. This requirement is to restrict unreasonable growth of the SIS. The intention is to encourage the use of libraries of reusable, portable software packages rather than continually increase the size of the SIS.
2. The item must not be implementable in terms of more primitive functionality without compromising one of the following:

Efficiency of implementation on some desirable APSE host.
Security characteristics.
System Integrity

The purpose of this rule is to strike a balance between the desire for a very primitive interface standard, thus decreasing the effort required to rehost, and a very high level interface, to try to ensure that all possible functions have been included.

3. The item should be required by more than one tool (or a small set of related tools.) The rationale for this rule is that if only one tool needs an interface, then that interface should be a part of a presumably small host dependent portion of the tool. For example, only the command line interpreter needs to log the user off, therefore the SIS should not force every host to implement a specific interface for logoff.
4. The item should not severely impact the rehostability of APSEs. There is a tradeoff to be made here. How important is the tool? How universal is its use? Can it be redesigned such that it does not use the functionality that is impacting rehostability? Can it be put into something other than the 'Basic SIS', for example? When an item passes the above criteria then a determination is needed for which SIS category it should be placed in. Most important, an item should be placed in as high a category as possible. For instance, it should only be placed in the Basic SIS if one cannot do reasonable Ada program development without it.

Conclusion

In conclusion, we should not lose sight of the motivation for the portability requirement: AVAILABILITY of rich Ada environments. One

3G-5

KAPSE Support for Program/Terminal Interaction

Fred Cox
Georgia Institute of Technology

Abstract

Programs with high-quality user interfaces commonly require extensive control over the input and output (I/O) of data to terminal devices. Interaction between programs and terminals is mediated by the operating system. Different operating systems handle terminal I/O differently, cause varying side-effects to the data streams, and provide software with differing degrees and kinds of control over this type of I/O. The Kernel Ada* Programming Support Environment (KAPSE) must provide software with full control over terminal I/O, while insulating the software from the quirks of individual operating systems. These requirements must be supported if the quality and portability of programs in the Ada Programming Support Environment (APSE) are not to be seriously degraded.

Introduction

In an interactive computing environment, the terminal is the principal physical component of the user interface. The quality of this interface depends heavily on the functions provided by the terminal and their control by software. The means by which terminals are controlled and the degree to which that control is available to applications software varies considerably from operating system to operating system and from terminal to terminal. This lack of standardization has led to low-quality user interfaces for most software and to low portability for much software providing high-quality user interfaces.

There is a popular diagram representing the Ada Programming Support Environment (APSE) in terms of concentric circles of increasing abstraction and decreasing physical detail. This diagram seems to place the user interface at the outermost locus. In fact, the user is physically interacting with a terminal device, which is physically connected to the computer hardware and controlled through the operating system. However, these components are represented in the center of the diagram. A program operating at or beyond the Minimal Ada Programming Support Environment (MAPSE) level must communicate with the user via these inner regions. This implies the user is located at the center of the diagram rather than outside it. It is the information transmitted through the terminal that permits the user to view the APSE logically from the outside of the diagram. The quality of the user's view is therefore dependent on the quality of interaction sustainable through the terminal device.

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

The building of good and portable user interfaces is complicated by differences in the way operating systems handle the synchronization, modification and transmission of input and output data streams. The many types of terminals encountered also adds to the problem.

In this paper, we will first review common operating system implementation of input and output. Then we will look briefly at the I/O facilities provided by the Ada programming language. A discussion of requirements for supporting program/terminal interaction and related problems will follow. Finally, the key requirements will be listed and conclusions summarized.

Only character-oriented terminals are considered in this paper. Graphics display terminals, non-keyboard input devices (such as graphics tablets, light pens and mice) and other interactive I/O devices (such as for voice I/O) are not explicitly considered. Although requirements for supporting interaction with these other kinds of devices overlaps requirements for supporting character-oriented terminals, the additional requirements should be addressed by those seeking to define KAPSE interface standards.

Implementation of Input/Output

Characters received from a terminal are normally put into an input buffer until the buffer is full or a special character or character sequence is received indicating the termination of the current unit of input (e.g., a line of text). The contents of the buffer are then made available to the program.

Characters to be transmitted to a terminal are normally also put into a buffer. This buffer is passed to a system service routine which transmits the characters to the terminal along with special characters to control the display. For instance, special characters may be sent which cause an effect similar to carriage return on a typewriter.

A variety of other services may be provided by operating systems, as indicated by the following examples.

- Normally, any character received from a terminal (operating in full duplex mode) is echoed to the terminal to appear on the display.
- Input characters may be converted automatically to upper case by the system.
- The system may filter certain characters and character sequences from the input stream for its own use. For example, a Control/Y character may be interpreted by the system as a command from the user to abort the current process.
- The system may automatically transmit a character or character sequence to the terminal as a prompt to the user whenever a read action is requested by a program.
- The system may also provide options for the ordering of read and write operations.

The terminal and the host computer must transmit and receive data at compatible rates. They must also coordinate on the number of bits per character or block and the kind of parity used for error checking. Some terminals and hosts interpret certain character sequences as requests to stop or start sending data. This capability helps to prevent loss of data when the transmitting unit is sending data faster than the receiving unit can process it.

In some cases, the data link between the terminal and the host contains modems which may be controlled by the host by means of special control signals. For example, a smart modem might be directed to dial a phone number to establish the data link.

The Ada I/O Facility

Ada provides two levels of I/O, one operating at the abstract level of files, the other controlling hardware at a low level. Low level I/O might be used for control of direct memory access (DMA), for example. At the file level of I/O, Ada supports a very simple abstraction of input and output operations. Terminal I/O is usually provided through the file I/O abstraction.

Neither of these facilities is appropriate for general program/terminal interaction. Forcing the programmer to implement terminal I/O using the Low Level I/O facility might mean bypassing the KAPSE and much of the underlying operating system. This method would probably result in inefficient use of system facilities and conflicts with system operation and Ada file I/O. It would also make the programmer work at an unnecessarily low level of abstraction and with unnecessary complexity.

The high-level, file I/O abstraction is insufficient to support general program/terminal interaction. It provides no control over operating system induced side-effects to the data stream, such as synchronization anomalies due to buffering, and modifications such as the filtering, mapping or addition of characters. No control is provided over the associated data link characteristics, such as transmission parameters (e.g., rates, parity, number of start and stop bits) and link control (e.g., automatic dial; hang-up and faulty-link detection). Nor are terminal control operations (such as querying for terminal operability, querying for terminal type, and the sending and receiving of special escape sequences) assured of support.

The Ada file I/O procedures CREATE and OPEN have parameters for passing strings containing the file NAME and the FORM of the file (i.e., options for implementation of the external file). It is possible but inappropriate to use these strings to indicate that terminal interaction is desired. Passing information by string is dangerous in any event, since it defeats the strong typing of Ada. This form of information passing is also a poor way to specify the great deal of detail necessary for terminal interaction. More importantly, passing terminal I/O control information via the CREATE and OPEN procedures limits this passing to one occurrence (i.e., when the file is created or opened) and so does not support the dynamic requirements of such control.

Whether or not these string parameters are used to pass terminal I/O control information, IT IS CRITICAL TO APSE PORTABILITY THAT THESE STRINGS NOT BE PASSED TO THE UNDERLYING OPERATING SYSTEM FOR INTERPRETATION. Portability will be precluded because the interpretation of these strings will differ from operating system to operating system. This is a result of the general problem of naming resources in different operating systems. Each operating system has

its own method of resource access, its own required format for the information in the string, and its own manner of interpreting the requests and information encoded in the string. This method also fails to guarantee availability of the required terminal I/O control facilities.

These parameter strings must be considered part of the KAPSE interface and must be provided a standard interpretation from host to host. This requirement is critical to portability of software across environments.

The preceding criticisms of trying to implement terminal I/O control in a manner supporting software portability are not to be interpreted as a condemnation of the Ada I/O facility. Program/terminal interaction is a complex and dynamic problem due to the lack of standardization, the rapidly changing capabilities of terminal devices and the growing expectations of users. It would probably be inappropriate to address this problem directly within the Ada language.

Since general program/terminal interaction is not supported within Ada and is normally supported by operating systems in a non-standard fashion (if at all), this facility must be provided by the KAPSE to insure its availability and to insure software portability. The use of the string parameters of file I/O procedures to pass information for terminal I/O control should be limited (recognizing that it is neither sufficient nor appropriate for most such information) and standardized in the KAPSE interface.

Discussion of Requirements

Synchronization and I/O Streams

Many programs use input one line at a time. Even if a program asks for only a character at a time, the operating system normally waits until it can put a full line of input from the terminal into a buffer (or fill the buffer) before it returns the characters one by one to the program. The end of a line of input may be indicated to the operating system by control characters, escape sequences, or other specially designated characters in addition to the "carriage return" character.

For most programs, this line-oriented input buffering poses no problem. However, there are some programs which require each individual character as it arrives and which cannot wait for a buffer to be filled or a special character to be returned. Screen oriented text editors and some interactive graphics programs serve as examples of such programs.

In input mode, these programs typically must receive each character as it is input in order to update the screen. It is unacceptable to require the user to input a special character or character sequence after each primary, input character to cue the system to deliver the contents of the input buffer to the program. Some smart terminals can relieve the program of this burden, but dumb terminals necessitate program control of screen update.

Some command language interpreters handle command lines character by character, checking for correctness as a line is being input, or permitting in-line editing of the command line for error correction. Without the ability to acquire each input character immediately, such interpreters could not be produced.

The major objection to providing immediate acquisition of individual input characters is the inefficient use of machine resources associated with this operation. However, screen editors are considered to be more efficient of the human resource which the machine exists to serve. The availability of screen editors also seems to have a strong influence on user acceptance of programming environments. This is especially true for the more sophisticated users. Fortunately there is evidence of improvement in the efficiency of this kind of I/O, since a major computer vendor has recently announced hardware that handles single character input several times faster than previously.

Since it would be unreasonable to make it impossible to implement screen oriented text editors or advanced command line interpreters in the APSE, immediate acquisition of input characters must be provided. The Ada language definition avoids this issue and leaves the Ada programmer at the mercy of side effects arising from system buffering of I/O. This I/O facility should therefore be provided to Ada programmers through the KAPSE interface.

Of those programs which do not require single character input, some do have requirements for modifying the set of characters and character sequences which terminate a line of input. These terminators are normally kept in a table against which each input character is compared as it is received. The KAPSE should provide for adding to or deleting from this set of terminators. (A poor way to achieve single character input would be to designate all remaining characters as terminators. This would generate a lot of unnecessary overhead. Fortunately, the terminator table is usually too small to hold an entire character set.)

On most systems, characters are lost if transmitted by the terminal before a request for input occurs. However, some systems provide a "type-ahead" buffer to capture these characters. Although this facility improves life significantly for the experienced user, it makes the user/program interface somewhat more complicated. If the KAPSE provides a type-ahead facility, it should also provide programs the ability to dynamically enable and disable the facility.

Another side-effect on I/O is caused by the method used to queue I/O requests. If all requests are put into the same queue, reads and writes are executed in the same order as they were requested. However on some systems, read and write requests may be put into separate queues. In this way, a write request which was queued after a read request might be executed before the read. Although this facility offers great potential for confusion, it also makes it possible for a program to output a lot of data (e.g., dumping a trace file) while watching for a signal from the user to desist (without aborting the program). Related to this queueing technique is the "break-through" facility. Whereas the double queue system may permit only complete reads or writes, the break-through permits a write to interrupt other I/O operations in progress. In this way, an urgent warning message may be displayed without waiting for other I/O activity to complete.

Both the double queue and the breakthrough facilities are implemented within the operating system. Therefore control of these facilities should be provided through the KAPSE interface.

Another feature commonly provided by operating systems is the choice of synchronous or asynchronous I/O. While there may be some reasons for giving programs direct control of this facility, it should not be required since the same effect can be obtained using the Ada tasking facility.

Modification of I/O Stream Contents

It is common for operating systems to modify the contents of input and output data streams, often in ways hard to identify. Stream elements such as characters or sequences of characters may be added to a stream or filtered from it as it passes through the system. Some elements may be converted into other forms by the system.

Input streams may have blanks, form feed or other extraneous characters added to them. Conversely, certain control characters and escape sequences are normally filtered from the input stream and interpreted by the operating system as signals, for instance, from the user to abort a process or to stop or start transmission of output data. Many systems automatically map lower-case characters into upper-case or perform code conversion (e.g., from ISO ASCII to CDC's 64 character set ordering).

To the output stream, operating systems commonly add character strings for prompting a user for input. If the terminal link is operating in a full-duplex mode, the system will echo input characters, putting them into the output stream. Some systems which keep track of terminal display width and the number of characters transmitted per line will add "carriage return" and "line feed" characters to provide a "wrap-around" effect at the terminal so all the data will be displayed. Line terminator characters in a file may be mapped to other characters or character sequences appropriate for a terminal when the file is being copied to the output stream.

In general, there are good reasons for each of these I/O stream modifications. However, they are very dependent on the particular operating system being used and consequently can have a serious impact on the portability of programs and data. At times, these modifications cause real problems for the programmer. Unwanted additions of characters to a stream can be difficult to predict and handle. Some programs need to get all input characters without losing any to system filtering. For instance, a program updating sensitive files may require that it not be aborted at will by the user, but be permitted to take remedial action before termination. Such a program will require any control characters the operating system would normally filter and interpret as abort commands to be passed through by the system to the program for interpretation. Another example of software requiring the pass-through of such control characters is EUNICE, which implements UNIX under VAX VMS. The testing of an operating system or command line interpreter (perhaps being developed for a different target machine) may also require all characters to be passed through to it.

While it may simplify the writing of programs such as command line interpreters to have all input characters automatically mapped into upper-case, other software such as word processing systems are severely handicapped by this mapping. Software documentation is degraded in quality when forced to be in all upper-case letters. Yet, it should be possible to develop documentation on the same system as the software it documents.

As regards modifications to the output stream, there are circumstances when a programmer needs to suppress any system supplied prompt string or to substitute his own. The echo facility also must be suppressed at times. It is common for operating systems not to echo the password when a user is logging onto that system, in order to protect the password. However, some applications programs also require the use of a password and need to be able to suppress the echoing of that password for the same reasons.

While not wanting to deny an operating system the ability ever to make the modifications mentioned, the programmer at times requires the ability to prevent these modifications. Since the kinds of modifications vary widely from operating system to operating system, the KAPSE must provide not only the capability to suppress these modifications but a uniform view of them as well.

Transmission of I/O Data Streams

The contents of I/O streams have to be physically transferred between the computer and the terminal. This transmission is characterized by such aspects as transmit rate, receive rate, kinds of error detection and correction implemented, number of bits transmitted per character or block (e.g., start/stop bits), synchronous or asynchronous interaction, duplex mode, and others depending on the protocol used. These characteristics may change during the period of program/terminal interaction.

Software frequently must be able to determine and modify these transmission characteristics. When I/O streams are transmitted across telephone links via modems, software may require control of automatic dial facilities, modem and link test capabilities and detection of hang-up or broken link states (e.g., to provide automatic logout or suspension of a process and reallocation of I/O devices). The KAPSE should provide facilities for determining and controlling these transmission characteristics and data link devices.

Terminal Control

Many terminals available today support a wide range of functions for controlling the display of information. Most CRT terminals permit positioning of the cursor, forward and reverse scrolling of the displayed data, selective erasure of parts of lines and the screen, and choice of characters sets (e.g., U. K. , or ASCII) and graphic rendition (e.g., underlining and reverse video). Some terminals provide further functions such as graphics character sets, raster graphics, automatic wraparound, various character sizes, saving and

AD-A141 576

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM PUBLIC REPORT VOLUME 3(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 25 OCT 83

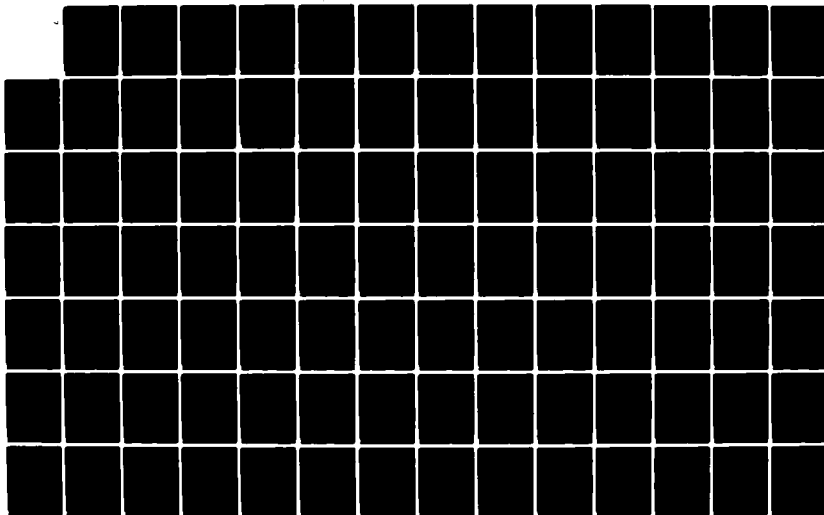
3/5

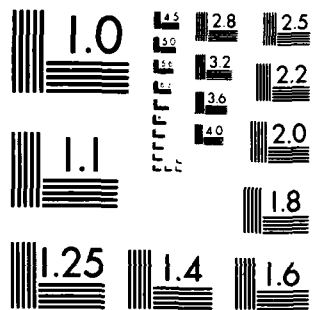
UNCLASSIFIED

NOSC/TD-552-VOL-3

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

restoring the cursor position, smooth scrolling, split screen display, tabulation, and self-test.

These terminal functions are invoked by sending escape sequences to the terminal. Some terminals are also able to identify themselves and their attributes, settings and status on request. This information is very important in controlling the interaction of a program with the terminal and is sent to the computer also encoded as escape sequences.

For instance, if the terminal supports the ANSI defined advanced edit functions (e.g., inserting/deleting a line or inserting/deleting a character in an existing line), writing a screen editor is greatly simplified and single-character I/O may not be needed. Other terminals may provide the "new line option" of generating carriage return linefeed characters when the "RETURN" key is depressed and interpreting a received linefeed character as a carriage return linefeed sequence, thus changing the control codes a program must generate. If the XON/XOFF communication protocol is supported, data transmission may be able to proceed at a higher rate and without fill characters (inserted to provide time for the terminal to execute its functions before receiving more data). Additionally, a given terminal may be capable of emulating several other kinds of terminals, permitting the program to switch the terminal's mode of operation to something convenient rather than having to generate special controls for each particular terminal. Most terminals are also able on request to return an indication of whether they are functional, thus providing greater control of exceptional conditions to the software interacting with the terminal.

Given these requirements for determining and controlling terminal characteristics, the KAPSE must ensure that the escape sequences generated by both software and the terminal can pass through to their proper destinations, unimpeded by the operating system. With the great and increasing variety of terminals available, it would be appropriate to provide an abstract terminal facility, probably at the level of the MAPSE. The objective in providing such a facility would be to permit programs to be less dependent on particular terminal types and therefore able to run effectively in a wider range of environments.

List of Requirements for Supporting Program Interaction with Terminals

1. interpretation of Ada file I/O procedure string parameters NAME and FORM by the KAPSE (not the underlying operating system) in a standard fashion
2. immediate acquisition and output of individual characters by the program (single character I/O)
3. specification of the set of characters or character sequences which indicate termination of an unit of input (e.g., carriage control characters and escape sequences)

4. control over type-ahead facilities
5. control over the order of servicing of read/write requests
6. provision of a break-through facility
7. control over the addition of characters to the input and output data streams (e.g., formfeed characters added to the input stream, and character sequences added to the output stream to effect wrap-around or to provide prompts)
8. specification of the sets of characters or character sequences which may be filtered by the system from input and output streams
9. control over modifications of characters in the input and output streams (e.g., mapping from lower case to upper case)
10. control over the echoing of input characters back to the terminal
11. dynamic control of host transmission characteristics (e.g., rates, parity, number of bits, full/half duplex)
12. dynamic control over the data link (e.g., for self-test, automatic dial, hang-up or broken link handling)
13. unimpaired transmission of control sequences and information codes between program and terminal (e.g., escape sequences for determination of terminal type and settings)
14. uniform view of all terminal I/O facilities from KAPSE to KAPSE

Conclusions

Since general program/terminal interaction is not supported within Ada and is normally supported by operating systems in a non-standard fashion (if at all), this facility must be provided by the KAPSE to insure its availability and to insure software portability.

The use of the string parameters of Ada file I/O procedures to pass information for terminal I/O control should be limited (recognizing that it is neither sufficient nor appropriate for most such information). These strings must be interpreted by the KAPSE (not the host OS) in a standard fashion.

The KAPSE must provide a standard set and a uniform view of functions in this facility. Programs must be able to disable certain features such as prompting, echoing, filtering, and line buffering. Software must be isolated from host operating system side-effects and peculiarities by the KAPSE.

These requirements must be met to insure portability of software and to support high-quality user interfaces.

In addition, it would be useful to provide a terminal abstraction facility in either the MAPSE or KAPSE.

Only character-oriented terminals have been considered in this paper. Graphics display terminals, non-keyboard input devices (such as graphics tablets, light pens and mice) and other interactive I/O devices (such as for voice I/O) have not been explicitly considered. Although requirements for supporting interaction with these other kinds of devices overlaps requirements for supporting character-oriented terminals, the additional requirements should be addressed by those seeking to define KAPSE interface standards.

THE DIFFICULTY IN DEVELOPING AN Ada ENVIRONMENT FOR BOTH RUN-TIME AND PROGRAMMING SUPPORT ENVIRONMENTS

Edgar H. Sibley
Alpha Omega Group

1.0 Introduction

The normal way of explaining the relationships between the hardware-and-"operating system" core of the machine, the KAPSE, MAPSE, and APSE is by means of a "target" diagram, with this core as the bullseye and the KAPSE, MAPSE, and APSE radiating rings. This model will be used here, but the semantics of this model must be defined, because they have been assumed to be "obvious" by most writers. However, these same writers have then used their own (non consistent) interpretations of the terms. Moreover, the terms KAPSE, MAPSE, and APSE tend to be interpreted in different ways by different writers, thereby compounding the problem. We shall not attempt to make any rigorous definition of any of these terms. Indeed, the provision of an exact definition of a KAPSE -- other than stating that a given implementation is an example of a definition -- would be a major research-and-development effort for several persons working for some time (say fifteen people for four years). Also, no definition such as this, with substantial complexity, is likely to be proven unless there is a validation procedure that may be applied to any implementation.

Kernel Ada Run-Time Support Environment (KARTSE) is a term coined to be similar to "KAPSE" but to deal with the minimal (kernel) run-time support environment that will be needed to support Ada programs. The RTSE is needed once a program has been developed on a PSE. An RTSE is normally supplied as a part of the Operating System (OS), but not all "target systems" come equipped with one. This is because the Ada environment was originally deemed to be one where the object code ran on the target system, which may have much less than a normal operating system today...possibly the bare-bones that allow several special programs to run in a minimum hardware configuration. Thus the PSE may be substantially different from the RTSE and a normal OS. When larger systems that are normally programmed to use the same host and target computer start to utilize an Ada framework, it will probably be necessary to reconsider the relative roles of the PSE and RTSE.

A mechanism for developing a set of requirements for a KAPSE and KARTSE that are effective in their interactions could be as follows:

First, let us assume that the KAPSE and KARTSE are relatively "empty" but that we start to fill them with all functions that are absolutely essential. This implies that we have a definition that is acceptable (to the KITIA, if not to the entire community). This is a first pass at the specification of an architecture of a part of the KAPSE. The specifications considered here are those needed to support the data and program library functions.

Second, we look at the run time support environment to see what must be added to (or maybe subtracted from) a KAPSE in order to allow it to be an effective "Kernel Ada Run-Time Support Environment" (KARTSE). Again, this is mainly considered in the area of data and program management.

Third, consideration is given to the problem of moving a program and test data etc. from a KAPSE to the KARTSE.

2.0 The Semantics of a Target Diagram

It may be assumed that there is a reason for a set of rings in a "bulls-eye" or "target" diagram. The author believes that such an arrangement is meaningless unless there is isolation between the rings. Thus an action at one ring interface can only be interpreted at the single ring under it, and no other. In the model presented here, the number of the ring increases outward and the interface designated $x-(x+1)$ is between level x and level $(x+1)$; naturally, the $x-(x+1)$ interface is the same as the $(x+1)-x$ interface.

The prime rules will then be:

Rule 1.

The only way that a function in one ring can interact with a function in another ring is through the interfaces between them,

using the standardized protocols that are defined across these interfaces.

Thus ring-1 communicates with ring-2 through interface 1-2, which requires that the communication be expressed in a well defined (local standard) language (possibly at a low level, not intended for human understanding). Also, when a function in ring-3 communicates to a function in ring-1, it issues its requests at the 2-3 interface and is no longer able to affect the outcome of the action; i.e., the resulting reactions at the 1-2 interface are hidden from and unavailable to the 2-3 interface (and hence the initiator function at the ring-3 level).

The corollary to this statement is that it is not possible to "drop through" from one level to another unless there is a "unity function" that expresses the relationship between initiation at one level and initiation at another level. This means that a lower level is totally hidden to the using function or person, except through functionality provided at the interface.

As examples:

1.a. Let us suppose that a function (c1) at level 1 is initiated by giving it parameters "a" and "b" by issuing the command at the 1-2 interface:

c1(a,b)

Let there be a function (c2) that will cause initiation of c1 by giving, at interface 2-3, the protocol:

c2(a)

In order for this to be possible, there must be an action at the intermediate level (2) that transforms the command "c2(a)" to its equivalent at the lower level "c1(a,b)." Thus, in level 2, the command c2(a) is validated and the additional parameter (b) attached. It is necessary for the system at this level to add the extra parameter (such as user class) so that its form will be valid at level 1. Note that the using function at level 3 does not have enough information to provide the extra parameter, because it is "hidden" at that level; even if the "user" had this knowledge, any call of type c1 would be invalid at the 2-3 interface or a call of "c2(a,b)" would have the wrong format at the 2-3 interface.

1.b. In the case where the commands have the same form (e.g., c'2(a,b) replaces the previous call at the 2-3 interface) then the command must still be checked for form before being passed, essentially unchanged (except for its "function name" being replaced by c1 instead of c'2).

Such isolation is essential in order to allow changes at the lower levels without affecting an upper level. This is a "level isolation" concept.

Rule 2.

The implementation of the set of functions that may be invoked at

any level may allow interconnection between any parts at that level and their implementation may produce further interface rings, but the architecture that results must not violate the first rule. Also, any additional interface created within this ring is there for the convenience of the implementor of the level (maybe a tool builder), and its addition does not imply an addition to the set of standard interfaces.

The implication of this rule is that a function (which may be a tool) at a specific level may interact with other functions at that same level, thereby providing a "suite of tools" that interact, but that these tools may not interact in such a way that they invoke another function at another level, except through the "standard" interface protocols. Moreover, the set of functions may provide the equivalent of an additional layer (e.g., interface A is introduced into 2; the parts of 2 are then 2.1 and 2.2; the interfaces 1-2 and 2-3 are unchanged, except that they may be designated 1-2.1 and 2.2-3 respectively, while interface A may be designated 2.1-2.2). However, the new layer is not defined anywhere except in the particular tool implementation.

This rule is important when considering the problems in specifying a protocol for inter-tool communication. Indeed, the implementers are apparently the only people that can assure that the interfaces are well defined. It is apparently impossible to have inter-tool interfaces that are consistent unless they are:

- (a) always at the same side of an interface and always communicating through the interface in order to be able to assure uni-

form protocol; or

(b) designed according to some well defined intra-layer standards. These "local standards" apply only to a single ring, and all tools must adhere to these standards in order for them to communicate correctly. Manufacturer specific or industry voluntary standards could be defined; then, any additional tool that adhered to the standard would be "mutually" portable. Such tools would not necessarily be compatible with other manufacturers tools, but their interfaces would be compatible with any other tool using the same standards.

It may well be that use of this rule will allow better decisions to be made on the location of the major software tools, such as a data resource dictionary, configuration management, and database management modules.

3.0 Data and Program Support in the KAPSE

Conventional DataBase Management Systems (DBMS) do not deal well with bulk data in either the form of the so-called unformatted file or of libraries of non homogeneous data, such as a set of programs that have been partially or fully link edited as a system ready to be executed. Such data occurs in streams of bits representing machine structures: words, bytes, paragraphs, syllables, or blocks, etc. The stream or its parts may be directly or serially accessible. Such data is sometimes called unstructured data, and the system is said to have no knowledge of the

internal form of the data. This may be true in some cases, but it does not capture the essential difference. For example, in Ada, a file is said to be "associated with an unbounded sequence of elements, all of the same type". It can be argued that the system is required to know the element type, to ensure that all users access it using the same element type.

A suitable treatment of bulk data is essential in the KAPSE, because the entities that are controlled through a Programming Support Environment (PSE) are primarily associated in storage as bulk text (e.g., Ada source, compiled objects, documents, text, and test data). In the past, the normal way to deal with bulk data has depended on its usage. If it was an entire system, a program, or a part of a program, etc., it was placed in a "library" that could access the unit by its name. The structure of the unit was generally simple or non-existent. If the data was to be accessed by a procedure, then the data was stored as a relatively conventional set of records in a file or in some similar fashion (indeed, it could be stored as a stream of characters or even as a stream of bits, but the procedures and the supplied access methods were the only way that the data structure was known). One of the special APSE data structures is, of course, the Diana tree. This has a structure that has been standardized, and the fact that the Diana tree has needed to be made a standard is an interesting example of the need for intra-level standards — in order to allow inter tool action within a level. Moreover, the fact that some designs today have problems with the compiler and related tools (e.g., the editor for Diana trees and the validation that is needed to allow this) suggests a greater need for adherence to the rules in the previous section; it also

leads us to ask whether it was really sensible to look on compilers and some other tools as a part of the MAPSE rather than the KAPSE layer, because they provide interfaces that may violate security.

3.1 The Data Support

As already discussed, there are at least three classes of data that are important in the Ada environment. These are:

(a) Unformatted data; this is a broad class of data that may have structure, but which has no structure that may be known to any program or procedure other than through special communication from a programmer or through a package and its relevant operators (in the sense of abstract data types). This class of data could be a report in character form (possibly internally indexed, or part of a word processing system with a retrieval mechanism), or it could be a traditional file (with a well defined file structure that needs a special access method -- such as ISAM -- to act as an indexing device for rapid retrieval), or a "bucket of bits" which may be the results of a transmission or a program. All types within the class of unformatted data have at least one characteristic in common -- they consist of a stream of bits that may have structure, but this structure is unknown outside the suite of procedures that access the data.

(b) Standard Formatted Data; such data has a predefined format that has been previously defined by a community-of-users that use a boy-scout method to ensure that all tool-using devices are doing so consistently. Typical of these data are the groups of

procedures that work on a common data structure. In a run-time environment these may be a suite of personnel programs that provide accounting, payroll, and personnel support services, while the compiler and editor interactions (via Diana trees) is an example of these types of systems in the APSE. The difficulty with such systems is that they rely on the good will of the users (or the hard heads of auditors) and often are violated, either deliberately or in error.

(c) Fully Formatted Data; this is seen in any database managed system and in some structured Programming Support Environments (PSE). The essence of such data is that the environment is aware of the structure of any data or part of the data. As an example, the PSE being designed in the UK has a built-in structure that assures that the program parts are properly controlled -- thus the subprograms of a main program are associated with it, and moreover the particular version of a subprogram that is valid with the particular version of the main program is associated properly. Thus the "structure" of this data is an exact match of the configuration that must be managed. Then the run-time environment will be able to retrieve the required version of the main program and with it all relevant and correctly versioned copies of its subprograms. Of course, the abstract data type concept also allows this, provided that the package definitions are available to other programs and known by the programmers.

3.2 The Program Support

In order to support the programming environment, it is necessary to have certain types of tools. These are categorized here as:

(a) Higher to Lower Level Language Translators; these are often considered to be compilers, but in the Ada environment they are split into parts -- a compiler from Ada language statements to a Diana tree representation followed by a low-level code generation from the tree to a target machine code program.

(b) Editor, configuration, and similar support procedures; these are some of the tools that make it possible to enter and alter language statements (and possibly change the Diana tree), to keep track of program versions, and to generally provide a good work environment for programmers.

(c) Program Library Support; which generally involves a device to catalog and store the programs and procedures. As discussed in the data support above, this library may be structured to provide one type of configuration management, or this may be provided externally as discussed later.

4.0 KAPSE and KARTSE Interaction

The run-time support environment (RTSE) and the programming support environment (PSE) are not really easy to differentiate, possibly because the PSE is really a particular type of RTSE; yet an attempt to do so for the programming environment has led to the partial definition of an entity termed a KAPSE. Since the Stoneman document was approved, it appears that the Ada community has forgotten that the prime reason for a programming environment is to provide programs that can be run -- presumably in a run-

time environment. The idea of a host-to-target environment, of course, has led to some of this apparent neglect, but most target machines have a need for some environment, and it seems reasonable to assume that future target machine architecture will benefit from the definition of a standard run-time environment. Naturally, any other run time environment that interacts with other machines/computing devices will benefit even more if a standard KARTSE exists.

Much then has been said of the KAPSE, but little of the KARTSE. If, as appears very likely, the Ada language is used to implement logistic and other large scale "non-operational" (large scale administrative) systems, then some of the requirements of a KARTSE that were only marginally necessary will be more obviously essential. These include the need for security features, a way to store data structures and define the meaning of the data entities (an information resource dictionary), a means for storing and retrieving data based on these definitions (a generalised DBMS with good user interfaces for query, table generation, and reporting), methods for recording versions of requirements, program structures and their relationships to data and users (a good software configuration management system), and interfaces to the system documentation (which can be a part of the combined configuration and information resource management system).

The DBMS-like features should include functionality that allows interfaces to a dictionary. Modern dictionaries have many different features, but they are all generally able to capture compiler data to document it and usage by programs. Some are even able to hold information on the users, security needs, and con-

figuration management controls. Any controls between a DBMS and a dictionary, and even the configuration manager, could be implemented through an "active " interface between the dictionary and its users (automated or human). The value of the dictionary is limited if it is passive -- merely recording the status; the value is fully realised when the system is active -- acting in a controlling role.

A short note on each of these features is now given:

1. Security Features.

Obviously, if the RTSE is a target machine that is not receiving any command signals after it is "started," then no security checking may be needed, other than at the start. And even then it may be unnecessary, because the target machine may be totally isolated (i.e., secure physically from outside influence).

If the target machine must communicate with other machines or devices, then some form of intercommunicational security may be needed. But if the target is similar to (or even the same machine as the host) or if the target has many external "users" (mechanical or human), then there is probably a need for some type of security.

There are at least three types of security that should be investigated: User Checking, Procedure Validation and Initiation, and Data Sensitive Checking.

User Checking is validation of the user (person, process, or

machine). It normally involves a check to see if the SIGN-ON mechanism is valid and some "hand-shake"/password means of validating the user. Generally, only one such validation is needed in a session.

Procedure Validation and Initiation occurs in two parts. The validation takes place during the promotion of the procedure from the debug library to the run-time library. This may therefore be considered a Software Configuration Management function. It entails the necessary checks that ensure that the program is fit to run. These are generally a matter of checking that any referenced data elements conform to naming and that debugging and other checking has been accomplished. The second part (initiation checking) is accomplished by a mixture of compile and run-actions.

This sharing of responsibilities depends on the architecture of the PSE and RTSE systems, because the question of whether the interaction of user and program is best handled as a user or program function, and whether the interaction of program and data should best be handled as a program or data function is architectural and performance oriented rather than design oriented. As an example, a certain user role may be allowed to utilize data from a particular file while applying the "Statistics-1" package, but never when the cells of computation have less than five data elements, and the local time is between 8am and 5pm, and this user has not made the same request or used the same set of data within the last 24 hours. In initiating this procedure, the type of input data that is valid is found at compile time of the Statistics-1 package, the matter of who, in general, may use any procedure or this particular package to access the particular

file may be a property of the file, and the rest of the restrictions must be placed on the request by the system at initiation time.

2. The Information Resource Dictionary

The term "Dictionary" has come to mean a set of automated procedures that aid all types of users by providing a mixture of normative naming, proper definitions, and reporting facilities dealing with the major entities (data, programs, and people) that interact in an information environment. The term "users" here means both animate and inanimate initiators of dictionary functions: e.g., systems designers, programmers, transaction processing clerks, query initiators, programs, and system libraries. The relationships between these entities may be relatively complex, and they are generally maintained so that the various users are able to interact in an easy way with the dictionary.

The "passive dictionary" has no major role other than to collect and report on the entities and their inter-relationships but an "active dictionary" will aid by controlling as well as recording. Thus the active dictionary will generally be able to provide at least some of the following:

(a) Collect data definitions from the programs and database definitions (Data Divisions and Database Definitions or abstract data type and their package definitions). Normally it is also possible for the system to both aid in generating these definitions and ensure (by use of a precompiler) that the data element naming, etc., in the programs is according to the standards

identified in the dictionary.

(b) Allow definition of validation procedures or consistency requirements so that the system can invoke them according to some triggering mechanism -- or at least include an automatic call to the procedure during the compilation when a statement is found to cause data to be input or changed, etc.

(c) Record the versions of the data and programs, and check that the versions of data definition and database are consistent with the version used in the programs (version compatibility). As an extension, the software configuration can be captured in the dictionary and some (possibly complex) controls enforced.

3. Software Configuration Management

A Software Configuration Management System (SCMS) is a manual or automated system that keeps track of the documentation and software of a complex information system development. Thus an SCMS would have references to all requirements documents, changes (proposed and resolved), all software developed, with test data and results, all relationships between the "version" of the programs, procedures, data definitions, etc. By careful use of the SCMS, a well documented and easily recoverable system is possible.

It is necessary to store this information somewhere, and as the data structures are complex in an SCMS, it is reasonable to

consider a DBMS as its storage device. Moreover, the DBMS can then provide the focal point of control. Thus one would expect that the SCMS in an Ada environment could provide a dictionary feature that is "active" and is able to ensure that the object code programs (or their data manifestation, the Diana trees) contain only valid dictionary names. In this case, the dictionary could be implemented as a special program using the DBMS, and the configuration management system could then be implemented as an extension to the dictionary — using the tools provided for dictionary extensibility as well as the procedural security and control provided by the DBMS. The testing of new programs, or of modifications would also be under the control of the SCMS, and no move of a program to production status would be allowed until the testing had satisfied all the SCMS conditions.

From the earliest days of programming, there have been attempts to retain and reuse large pieces of code; indeed, the early trigonometric functions and sort routines grew from such a need to avail of other peoples already debugged routines. Today, with software costs rising it is even more important not to keep on recoding the same pieces of code for different machines or for different purposes (but the same algorithm). Methods are now becoming available to allow systems to call out possible "chunks of code" to reduce the overall effort. The techniques again call for the ability to utilize complex relationships to provide a coverage for cataloging and indexing techniques for the classification of potentially reusable software. This may again be implemented as a part of an extensible dictionary.

From this discussion, it will be seen that the DBMS may be defined as a software package that exists in the KARTSE and has control over the storage of any data that has structure and longevity. Thus a string of characters, having structure, may be stored using the DBMS, and therefore the DBMS in a KAPSE/KARTSE will have to deal with textual message or compiled programs with their linked subprograms. It is clear that if a DBMS is to be provided for use by the PSE it would be very convenient and useful if the same DBMS were also suitable for applications use. This would mean that there was a DBMS in the KAPSE to allow for the addition of a formatted data concept to the PSE; as a result, the DBMS could be used for other functions and if the same DBMS were to support such actions as an ad-hoc query, then the KARTSE would potentially have the same interface DBMS.

5.0 Conclusions with a Proposed Architecture

This paper was written primarily to propose that there was a need to consider the use of the KAPSE as a KARTSE and to suggest some of the problems in transitioning from one to the other. It seems reasonable to look at architectures that have been developed for Operating Systems, Database Management Systems, and Information Resource Dictionary/Configuration Management Systems in the past. One possible high-level architecture is given in Figure 1. This shows how it might be possible to use the general architecture of the Stoneman KAPSE for a combined programming and run-time support environment, but some additional controls would have to be

added to the compilation process, and the software configuration management system would need to have control of any access to the libraries.

The development of a standard combined programming and run-time support environment for Ada would make future tools easily transportable and allow real "software reusability," thereby reducing the rising costs of software while allowing major systems of the future to be implemented in spite of the expected "gap" in available programmers and systems implementers.

DATABASES
WITH DATA STORAGE STRUCTURES
AND DATABASE SCHEMA

I

MACHINE HARDWARE AND
ANY OPERATING SYSTEM

I

I

ACCESS METHODS, IF NOT IN OS

I

I

DBMS...CHECKING AGAINST SCHEMA
DEFINITION ABOVE

I

I

MULTIPLE INTERFACES OF DIFFERENT
DBMS MODELS (COMPILE LEVEL)

I

I

QUERY LANGUAGE INTERFACES FOR
VARIOUS MODELS AND USER INTERFACES

I

I

I	DATA	I	SECURITY	I	CONFIGURATION	I	LIBRARY	I
I	DICTIONARY	I		I	MANAGEMENT	I	MANAGEMENT	I
I		I		I		I		I

KAPSE/KARTSE INTERFACE

APPLICATIONS AND COMPILERS, ETC.

NOTE: The lines in this diagram should be interpreted as arcs of
a KAPSE/KARTSE with the top as the bulls-eye or center.

Figure 1. An Architecture for a Possible New Environment

MINIMAL HOST FOR THE KAPSE

William L. Wilder
SofTech

Abstract

The concept of a minimal host for the KAPSE is explored and several criteria for categorizing the minimal KAPSE host are collected. This criteria includes the users' view of the environment that the KAPSE has to support, the Instruction Set Architecture of the host, the host's hardware configuration, and the host machine's operating system. Some recommendations as to the actual requirements for the minimal KAPSE host are given and several interesting conclusions about the minimal host for the KAPSE are drawn.

Introduction

As described in the "Stoneman" document, the Kernal Ada Programming Support Environment (KAPSE) provides services to the Minimal Ada Programming Support Environment (MAPSE) that allow the MAPSE to execute on the host computer and that isolate any host dependencies from the MAPSE. The MAPSE provides the program generation facilities that allow Ada applications to be developed and then transferred to a target computer for execution. The MAPSE program generation facilities include compiling tools and database tools, among others, that must be written in Ada (per "Stoneman"). This means that a host computer for the KAPSE is also an Ada target computer, and hosting (or rehosting) the MAPSE requires a (re)targeting of the MAPSE program generation facilities and a (re)implementation of the KAPSE for the host.

The concept of a minimal host for the KAPSE would be useful in deciding upon an initial host for a MAPSE or in evaluating candidate host computers for rehosting a MAPSE. This concept would relate specific KAPSE functionality to the host computer's capabilities, and this relationship could determine what KAPSE functionality would be available to a MAPSE. The KAPSE for a minimal host includes both the services provided by the KAPSE and the implementation of the Ada language as defined by the Ada Language Reference Manual (LRM). The Ada implementation must be supported either by an Ada run-time system, or directly by the KAPSE, or by some combination of the two. Additionally, the minimal KAPSE host concept would begin the analysis necessary for designing new host computers for which KAPSEs will be developed.

This paper collects the necessary criteria for determining a minimal KAPSE host and categorizes this criteria. Some of the criteria are stated as recommendations, not requirements, because of the desire to develop a complete set of criteria for the minimal KAPSE host. Any recommendations in this paper concentrate on the criteria and offer only opinions as to what the requirements will be. These recommendations are based on the KAPSE implementations for the Ada Language System (ALS) and the Ada Integrated Environment (AIE), and studies on other possible KAPSE implementations. Research into the actual requirements for the stated criteria could then add the necessary level of detail that would provide a specification of the generic minimal KAPSE host.

Some terminology needs to be defined for categorizing the minimal KAPSE host. These definitions include both the hardware terms "byte" and "word", and the software terms "Ada program", "process", "task" and "subprogram". For the purposes of this paper, a "byte" is eight bits of binary information and a "word" is four bytes (or 32 bits) of binary information. An "Ada program" is any compiled and linked entity of Ada source code that executes on a host computer using KAPSE supplied services. In this paper, the term "process" refers to an executing Ada program. An Ada program may be executed in several concurrent instantiations, resulting in one process for each execution. The terms "task" and "subprogram" are defined by the Ada LRM and are used only in that context.

Categorization of Criteria

The criteria for a minimal KAPSE host will be categorized according to the users' view of the environment that the KAPSE has to support, the Instruction Set Architecture (ISA) of the host machine, the host's hardware configuration, and the host machine's operating system. The criteria for the user's view of the environment are based on various "Stoneman" requirements, while the ISA criteria are based on the Ada language. The criteria for the hardware configuration are a combination of top-level "Stoneman" requirements and implementation-level KAPSE requirements. The host machine's operating system criteria are a combination of the KAPSE requirements and the Ada language.

The users' view of the environment is really their view of the MAPSE associated with the KAPSE. Therefore the criteria for this categorization will be for the MAPSE that must be supported by the KAPSE. Some observations on the ISA will indicate possibilities for implementing various Ada language constructs. The necessary capabilities for the Central Processing Unit (CPU), physical memory and address space, I/O channels, online and offline storage, and various peripherals will be outlined for the hardware configuration. These capabilities are given at both a minimum recommended range and a more reasonable level.

The three possibilities for utilizing the host machine's operating system in implementing the KAPSE are: (1) implement the KAPSE on the host machine without the operating system, (2) implement the KAPSE on top of the operating system, or (3) implement the KAPSE beside the operating system. With the first alternative, only the user's view of the environment, the ISA, and the hardware configuration need be considered in categorizing the criteria for the minimal KAPSE host. When implementing the KAPSE on top of the operating system, the KAPSE should attempt to utilize existing operating system functionality. The approach of implementing the KAPSE beside the operating system can be viewed as some mixture of the first two alternatives. The last two alternatives of implementing the KAPSE on top of or beside the host machine's operating system will be discussed in parallel (see the Operating System section).

Users' View of the Environment

The user's view of the environment that the KAPSE has to support must be determined before the first criteria for the minimal KAPSE host can be explored. This view of the environment can be broken down into the areas of multiple-user/multiple-tool systems, single-user/multiple-tool systems, and single-user/single-tool systems. The "user" portion of each system characterization refers to the number of interactive MAPSE users that are active at any time, while the "tool" portion of each system characterization refers to the number of Ada programs that can be executing at any one time. Some examples of these systems are: UNIX, which is supported as a multiple-user/multiple-tool system on numerous computers; and personal computers, most of which can be classified as single-user/single-tool systems. Although no particular systems are discussed, this criteria for the minimal KAPSE host will be based on existing systems, expected enhancements to these systems, and new systems that will be available in the foreseeable future.

The multiple-user/multiple-tool systems are recommended for the minimal KAPSE host because it is generally accepted that the MAPSEs will be directed toward these systems. Note that the KAPSEs currently being implemented for the ALS and the AIE are supporting MAPSEs for multiple-user/multiple-tool systems. It is assumed that the distributed-user/multiple-tool systems of the future will be an outgrowth of these multiple-user/multiple-tool systems as the distributed MAPSE comes of age. The single-user/multiple-tool systems are the lower bound of the multiple user/multiple tool systems and probably not hosts for any currently planned MAPSEs, but these systems could become the minimal KAPSE hosts of the future when programmer work stations become part of the MAPSE. The single user/single tool systems will not be considered because these systems are not expected to ever host a MAPSE.

Instruction Set Architecture

The Instruction Set Architecture of any machine impacts the Ada language implementation in several major areas: (1) the machine's instruction set; (2) the addressability of code and data; (3) the architectural facilities for reentrancy, recursion, and tasking; and (4) the hardware support for

eptions. The instruction set of the machine should be comprehensive enough to ensure that implementation of the Ada language constructs do not cause any serious time or space penalties. The generated code should be reasonably obvious; it should not be necessary to make complex choices between alternative means of achieving similar effects or to contort the code in order to exploit specialized instructions. The machine's instruction set should allow for referencing data fields of various lengths--bits, bytes, words, and multi-words.

The techniques for allocating space to declared objects in Ada subprograms, and for their later access, should be straightforward. The use of special instructions dependent upon the location or alignment of data should be minimal. Data addressing should be simple; a direct addressing scheme in which there is no need to use base registers and/or indexing in every reference is useful. Ideally, both direct addressing and offset addressing capabilities should be recommended so that the implementation of any Ada language construct could use either (or both) addressing schemes as required. When offset addressing is available, there should be no need for an elaborate strategy in allocation of base or index registers when addressing data. Any arithmetic required for these addressing purposes should be straightforward.

Code may be executed in a reentrant manner by the various tasks within a single Ada subprogram (i.e., tasks can share both code and data); therefore the generated code must be reentrant. On many MAPSEs, it will be desirable to compile Ada programs (especially large programs like the compiler) so that the code can be shared by several users. Any Ada implementation requires the handling of block-structured data areas because Ada is a block-structured language, and, potentially, all Ada subprograms can be recursive or can utilize tasking. This means that the capability to reference code or data via offset addressing is recommended for implementing reentrancy, recursion, and tasking. Some form of memory protection (probably along task boundaries) would be helpful in isolating the various data areas. Numerous stack or queue implementations are expected to be present in the generated code, and specific architectural support for stacks or queues would be a bonus, although many successful implementations have been done for other languages without this.

Hardware support for Ada exception handling ranges from those machines that offer useful assistance in the implementation of Ada exceptions to those that make exception handling very difficult. On some machines, the hardware interrupt structure provides a convenient mechanism to trap and possibly propagate Ada exceptions. This type of hardware support for Ada exception handling is recommended. On other machines, it is very awkward or costly to determine which of several possible Ada language exceptions should be raised.

Hardware Configuration

The hardware configuration must provide a powerful CPU, adequate physical memory and address space, numerous terminal and printer connections, sufficient online storage capacity, and offline storage capabilities. The processing power of almost any currently available CPU with instruction execution speeds in the sub-microsecond range (i.e., below 1000 nanoseconds) is recommended. The address space could be implemented by the operating system (or KAPSE on the bare machine) either directly in physical memory, or as a virtual memory space, or via overlaying. Terminal connections equal in number to that of the expected concurrent users are necessary in any interactive system, and several printer connections are desirable. Online storage includes various disk drives and possibly other direct access devices, while offline storage capabilities include mountable media (e.g., disk packs) for direct access devices and/or online tape drives.

A minimum physical memory size of 256 (2^{18}) Kbytes or larger is recommended, whether or not all of physical memory is directly addressable (i.e., having to use base registers and/or indexing for addressing). This translates into 64 (2^{16}) Kwords or larger of physical memory, but at least one Mword of physical memory (2^{20} words, 2^{24} bytes) would be better suited to the multiple-user/multiple-tool systems. A minimum address space of between 256 (2^{18}) and 1024 (2^{20}) Kbytes is recommended and could be available through actual physical memory or hardware support for paging (i.e., page faulting) in a virtual memory system. This translates into between 64 (2^{16}) and 256 (2^{18}) Kwords of address space, but several Mwords of address space (at least 2^{20} words, 2^{24} bytes) would be better suited to the multiple-user/multiple-tool systems.

Various configurations of peripherals, such as terminals, printers, disk drives, and tape drives, must either be supported directly by the KAPSE or be accessible through the KAPSE. Hardcopy and video terminals should be available and, if supported, the virtual terminal handler should be able to operate on the video terminals. When available, the printer must be accessible through the KAPSE to allow for the generation of listings. Disk drives with the capacity of approximately 100 Mbytes and transfer speeds of approximately 500 Kbytes/second are recommended for the minimal KAPSE host to keep the KAPSE database online. It is expected that disk drives with the capacity of 500 Mbytes or several disk drives with the combined capacity of a 1000 Mbytes and transfer speeds of over 1000 Kbytes/second would be better suited to the multiple-user/multiple-tool systems. When available, tape drives will be supported by the KAPSE to allow for file transportation and archiving.

Operating System

In this section on operating systems, the primary discussion will concern implementing the KAPSE on top of the host machine's operating system, with any additional discussion of implementing the KAPSE beside the host machine's operating system being in parenthesis. Any implementation of the KAPSE will have to provide for concurrent users and, depending upon the characteristics and utilization of the host machine's operating system, two distinct possibilities for the overall KAPSE organization can be visualized. On the one hand, the KAPSE could be organized as a single multiple-user subsystem that would be active whenever the KAPSE is being used and which could serve individual users. On the other hand, the KAPSE could be offered as an operating system (or KAPSE) process that could be invoked by the user through the facilities of the host machine's operating system (or KAPSE); thus at any time, there would be an executable image of the KAPSE for each active user.

While either of the above approaches is quite feasible, in practice most implementations probably will use a hybrid organization that combines aspects of both (e.g., a control process would be invoked for each Ada user, but this control process would use services provided by a single central subsystem). Within the context of this hybrid organization for the KAPSE, such areas as program execution, I/O support, and the KAPSE database will be addressed.

Program execution includes invocation of, termination of, and communication between processes associated with the executing Ada programs. I/O support addresses support for the SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO Ada packages by the KAPSE. Both program execution and I/O support consider the ramifications of Ada tasking on their respective implementations. The KAPSE database area includes the underlying file system of the host machine's operating system (or a disk area maintained by the KAPSE) and the KAPSE database access software.

Program Execution

Whenever an Ada program is executing in the KAPSE, certain basic facilities should be available in that program's execution. These facilities may be viewed as an extended run-time environment for the Ada programs provided by the KAPSE on the host computer. Program execution should offer the ability to invoke a specified program from within a running process; establish communications between the two processes, if possible; and return control and status information to the caller upon completion of the invoked process.

The basic recommendation for Ada program execution by the host operating system (or KAPSE) includes either (1) creating another process for the Ada program invoked and each process executing independently, or (2) having the invoked Ada program subsume the calling Ada program's process during the invoked program's execution, or (3) suspending the calling Ada program's process during the invoked program's execution. Some form of message passing between processes must be available at least during program initiation and termination, and full inter-program communication should be available if a suitable form of inter-process communication is supported by the host machine's operating system (or can be implemented within the KAPSE). Additionally, any implementation for program execution should permit processes to exploit the resources normally available to a individual process, rather than constrain the entire set of processes to share these resources.

When implementing on top of (or beside) an existing operating system, multi-tasking represents one of the most challenging aspects of the Ada language. There are two obvious approaches; either use one process for each task in the Ada program, or use one process for the entire Ada program with the

Ada run-time system implementing multi-tasking by multiplexing within that process. Taking the Ada LRM into consideration for tasking suggests that the process structure of the host machine's operating system (or KAPSE) is unlikely to provide all of the facilities that are needed for the Ada tasking model. Therefore most implementations probably would adopt the approach of multiplexing within a single process. The multiplexing approach, however, has its own potential difficulties in pre-emptive scheduling of higher priority tasks and in time-slicing of tasks within priorities. Also, a multi-task process implies multiple stacks, which must be isolated from each other.

I/O Support

I/O support must be provided for both sequential and direct access Ada files, as defined in the Ada packages SEQUENTIAL_IO and DIRECT_IO respectively, and for the production of human readable text as defined in the Ada package TEXT_IO. Since operating systems already provide these basic file access methods, it is recommended that the I/O for sequential and direct access Ada files should utilize the available operating system functionality. Unfortunately, tasking considerations may require that the KAPSE provide all I/O support (see the tasking discussion below). The production of human-readable text also utilizes sequential access Ada files and the host machine's operating system (or KAPSE) functionality, in addition to the actual KAPSE or Ada implementation of TEXT_IO.

Task switching should occur when one task of a multi-task program initiates an I/O transfer and suspending the entire process is undesirable. With a multiplexing implementation of tasking, however, such behavior may be difficult to accomplish. The host machine's operating system may suspend the entire process on the first I/O request or may refuse to accept later requests until the first transfer is complete (any KAPSE implementation should do neither). Where multiple transfers can be initiated, there may still be a problem with the maximum number of files or I/O streams that a single process can have open concurrently. There also may be some difficulty in awaiting completion of any one of several requested transfers and, in particular, the pre-emptive scheduling of tasks demands that notification of completion for an I/O transfer be given asynchronously.

KAPSE Database

There are two extreme approaches to the arrangement of the KAPSE database structure on (or beside) the underlying file system of the host machine's operating system. At one extreme, each file in the KAPSE database could be mapped onto a file in the underlying file system, so that there is a one-to-one correspondence between the KAPSE database structure and the files in the underlying file system. At the other extreme, the KAPSE database could obtain a large file from the underlying file system (or a single area of unstructured disk space) and the entire KAPSE database structure could be implemented in this large file (or disk area), relatively independently of the underlying file system. Various approaches falling somewhere between these two extremes are possible and most likely will be the implementation of the KAPSE database.

The recommendations for implementation of the KAPSE database access software are essentially the same as those of the overall KAPSE implementation. The KAPSE database access software may be implemented as one central process, or as a process instantiated by each individual user, or as a hybrid approach in which part of the database access software executes locally in the user's processes but communicates with a central process for some services. Again, it is expected that the hybrid approach probably will be adopted by most implementations of the KAPSE database access software.

Whichever of these approaches is adopted, potential problems will be encountered. A centralized approach must be able to identify individual users in some appropriate way for checking database access rights, while a localized approach must be able to maintain the overall consistency of the KAPSE database through synchronization of access. The hybrid approach could solve these problems by having the central process access the database on behalf of the user process to open the underlying file (or a portion of the disk area), and then have the user process read and write to the file (or a portion of the disk area). Unfortunately, many operating systems do not permit open files to be transferred from one process to another (any KAPSE implementation should). All KAPSE database access problems essentially arise from the need to permit access to the KAPSE database from within the MAPSE, while guarding against non-Ada access and achieving reasonable efficiency in the implementation.

Conclusions

The criteria for a minimal KAPSE host have been explored by their categorizations and several interesting conclusions can be drawn about the environment, the ISA, the hardware configuration, and the operating system. The initial user view of the environment for the MAPSE (and its associated KAPSE) are the multiple-user/multiple-tool systems and any minimal KAPSE host recommendations that have been specified in terms of these systems. Additionally, the MAPSEs of the future most likely will expand to include both the distributed-user/multiple-tool systems in a distributed MAPSE environment and the single-user/multiple-tool systems with programmer work stations; therefore the recommendations for the minimal KAPSE host will change.

The host computer's ISA should provide a comprehensive instruction set with sufficient data addressability and the necessary architectural features to support reentrancy, recursion, tasking, and exception handling since the Ada language must be available in any MAPSE. These ISA features directly relate to the efficiency of Ada programs and, therefore, the overall productivity of the MAPSE, but are not obvious criteria for minimal KAPSE host because, for example, a restricted instruction set could be offset by an extremely fast instruction execution speed of the CPU.

The critical components of any hardware configuration that should be closely examined in the minimal KAPSE host are: instruction execution speed of the CPU, physical memory and address space, and online storage capacity (including transfer speed). Other components (i.e., terminals, printers, and offline storage devices) of any hardware configuration are not as critical, but provide the necessary user interface to the MAPSE. The expandability of the hardware configuration should also be considered because, although it will not directly affect the criteria of the minimal KAPSE host, it does relate to how an individual host might evolve as the needs of the project or projects change. More main memory may be provided, the number of terminals can be increased, additional disk drives may be connected, and other peripherals (e.g., remote and teleprocessing devices) utilized.

The options available for utilizing the host machine's operating system when implementing the KAPSE are either to use no operating system at all or to implement in conjunction with operating system (i.e., either on top of or beside). Implementing the KAPSE directly on the hardware provides all the implementation freedom and problems of a bare machine. The ideas presented about the KAPSE implemented in conjunction with the host machine's operating system could be incorporated into the operating system functionality that must exist in a KAPSE for a bare machine. When implementing the KAPSE on top of or beside the host machine's operating system, the KAPSE should attempt to utilize existing operating system functionality and reduce the amount of implementation effort required. If the host machine's operating system does not provide the necessary support, the KAPSE must be able either to provide the functionality or limit itself to a restricted capability.

Acknowledgment

I would like to thank Patricia Oberndorf of the Naval Ocean Systems Command and Rich Thall of SofTech, Inc., who are also members of the Navy's KAPSE Interface Team (KIT), and Dr. John Gannon of the University of Maryland for reviewing a draft of this paper and offering many helpful suggestions.

References

Reference Manual for the Ada Programming Language, proposed standard document; U.S. Department of Defense; July 1982.

Requirements for Ada Programming Support Environment, "Stoneman"; Department of Defense; February 1980.

Ada Language System KAPSE B5 Specification, draft; SofTech, Inc.; February, 1981.

Ada Integrated Environment: KAPSE/Database Type B5, draft; Intermetrics, Inc.; June 1982.

FASP/ASP Ada Study; SofTech, Inc.; April 1982.

Ada Support System Study, Phase 3 Report "Support System Interfaces"; System Designers Limited; November 1979.

Ada Support System Study, Phase 4 Report "The Initial Host"; System Designers Limited; April 1980.

OF MICE AND COMMAND LANGUAGES : KAPSE INTERFACE SUPPORT FOR INTERACTIVE TOOLS

Roy S. Freedman
Hazeltime Research Laboratories

1. Introduction

APSE tools that can be controlled by a human require an interface to support the interaction between a device that the human controls directly and the tool that is to be controlled. If we want these tools to be transportable across APSEs, then KAPSE interfaces must be provided that support this interaction. This paper provides a rationale for a future set of draft KAPSE interface specifications for tools that interact with the human, through the KAPSE (and not directly through the host operating system). These tools (and necessarily, their interfaces) are supported by special interactive hardware devices. KAPSE tools that are characterized by their important interactive features are command language processors, editors, and symbolic debuggers. Other APSE tools that have important interactive components are mail tools, configuration control tools, and display tools. To permit transportability, these tools communicate with the KAPSE and with a device interface that may model a terminal keyboard, a terminal display (CRT screen), a light-pen, or a "mouse."

In this position paper, we are concerned with modeling the KAPSE device interfaces that support the human tool user. These interfaces are important in order to permit transportability of existing tools and to help future APSE builders develop newer tools that exploit more powerful (and less expensive) hardware functionality. Many of the ideas

in this paper were refined from an earlier draft with the comments of the members of KITIA Working Group 1, R.Thall, and R.Frall.

2. The Nature of Interactive Tools

Interactive tools interact with a human on a variety of levels. The characteristics of these levels have been detailed very carefully in [3]. A basic taxonomy shows that these tools can be classified as having "menu," "fill-in-the-blanks," or "parametric" human interfaces. All MAPSE and APSE tools mentioned above can be classified by this taxonomy. Newer tools and environments (for example, Smalltalk-80, Interlisp, Apollo Aegis, and Apple Lisa) exploit this taxonomy, by offering the human user a choice of various "modes" he would wish to interact. For example, a tool may initially assume a parametric mode, but may then switch to a menu mode if the tool user is regarded as "inexperienced" by some time-out mechanism. These environments frequently use several images or "windows" displayed simultaneously on a screen (for output), and use pointing devices for input in addition to the terminal keyboard. These features are not explicitly built in today's APSEs but are seen in other programming support environments. The Interlisp environment (referenced in section 4.D.4 on command languages and control in Stoneman) is an example of one such environment. All tools are completely integrated since the command language of this environment is the same as the target application language. Some implementations support process invocation in conjunction with window creation, so that a human user can see how his tools invoked other tools. This gives the illusion of performing several tasks simultaneously at "different" terminals. When coupled with an input device used for pointing (frequently a "mouse"), human

productivity is enhanced. Certain textual (keyboard) commands can be replaced by a menu mode, working in conjunction with push-buttons on an input pointing device. This device can also be used to facilitate the movement of windows. This further gives the human more control over his interaction.

This multiple windowing capability is available not only in Interlisp. Several powerful editor tools (in particular, EMACS) support multiple windows on a variety of terminals. These tools are hosted on a variety of programming environments. The AIM tool also supports a windowing capability, and is supposed to be transportable to both the ALS and AIE.

The ALS and AIE have the capability of supporting many features. The ALS provides for screen and text modes for the editor but this reflects the interface of the editor and the VAX VMS operating system. The ALS editor interface does not explicitly utilize the KAPSE, since it uses the standard VMS supported editor. (However, the use of this tool may be only temporary.) The AIE has an EMACS-like screen editor (without a multiple windowing capability). The device support for this editor is handled by the KAPSE interfaces contained in the terminal handler packages. The AIE designers are aware of possible modern extensions to their environment by allowing for a more powerful display capability. On page 115 in the B-5 for the AIE KAPSE (IR-678-2 November 1982) it is stated that

It is expected that the terminal handler will be enhanced to support multiple programs simultaneously on separate parts of the screen, with additional control characters for moving between the various screen windows.

The UK APSE also recognizes the need for highly interactive input-output. A "virtual terminal" package is to be provided for screen addressable tools. Again, this is similar to the AIE design, but no explicit KAPSE interface is provided for these tools.

These requirements are also found in section 4.D in STONEMAN, where several categories of interactive devices are specified, and where it is also observed that "the same control signals will be accepted by the terminal interface routines from all devices of these types." If these devices are to be supported in future APSEs, and tools that use these devices are to be shared between APSEs, standard interfaces must be provided in order to support "devices of these types." If these interfaces are not provided, transportability of tools that use these devices will be very difficult to achieve.

3. Device Dependent Interactive Tools

The Ada input-output package specifications `DIRECT_IO` and `SEQUENTIAL_IO` provide interfaces to "virtual" file-like structures. These package specifications are adequate for interaction that is textually oriented. These package specifications are good interfaces because their semantics are defined in terms of an abstract model (the "virtual" file). This existence of an abstract model for KAPSE interfaces is an example of the property of "capacity transparency," [1]. `DIRECT_IO` and `SEQUENTIAL_IO` have the property of capacity transparency because they are defined in terms of an abstract model of a file.

The other input-output package that Ada provides is too general and supports no virtual model of any input-output device. As observed in [2], the package specification for `LOW_LEVEL_IO` is totally inadequate to provide for many important KAPSE services that an interactive tool must utilize. This package specification relates to the input-output needs for an embedded application-specific target.

We now suggest some other abstract models of input-output that can be used to help express the semantics of KAPSE interfaces for interactive tools.

Most APSE tool builders define a KAPSE interface package called `TERMINAL_IO` or `TERMINAL_HANDLER` (or some other similar variation) that supports all interaction that a human tool user might desire from a non-file-like device. In order for this interface to be standardized, a "virtual terminal" or abstract model must be provided. The problems that APSE builders seem to be having in this area are that the different APSE builders use different abstract models for their "terminals." These problems are compounded by the fact that a terminal is treated as both an input and output device. In reality, not only are the terminal input and output functions logically distributed, but they are also physically distributed as well. It is suggested that these terminal input-output package specifications be functionally distributed into `KEYBOARD_INPUT` (emphasizing input) and `DISPLAY_OUTPUT` (emphasizing output).

As Fred Cox observed in [2], the keyboard input interfaces in both the ALS and ALE have much to be desired. A standard `KEYBOARD_INPUT` KAPSE interface would have to model single character input and line oriented input. Other requirements (that include character terminator tables) for such a standard are succinctly given in [2].

KEYBOARD_INPUT supports the typed human input associated with characters and text.

Typed human input can also be associated with addresses on a screen, either through the arrows (or control characters) of cursor control, or with the aid of a pointing device. This functionality can be accommodated within the package specification for KEYBOARD_INPUT by including inside it another package specification. This package specification (which we will call package MOUSE) will support devices that have the capability of addressing the screen, where the addressing mechanism is not necessarily associated with a keyboard. Bit-mapped addressing would be supported by package MOUSE.

Package MOUSE can be used to provide the KAPSE interfaces for the support of track balls, light pens, and conventional mouses (mice). One important requirement for the input facilities of this package is that a function be provided that senses the movement of the input device. This movement must also be coordinated with the display on the screen to take advantage of its interactive nature.

Audio input can also be provided in a similar fashion. A package specification can be developed to support a transducer or a microphone and would help support tools requiring audio input.

Package specification DISPLAY_OUTPUT can contain other package specifications that support the functionality (and granularity) of output devices. One specification can support the functionality of a teletypewriter. Another can support the functionality of a character-generation display, which has the capability of addressing characters on the display. This would support a (limited) windowing

capability. Another package specification can support the powerful functionality of a bit-mapped display, where individual pixels can be addressed. Another package can also be developed to provide for a color capability for all display terminals.

As observed above, audio output can also be provided by a package that supports the capabilities of a transducer (a speaker). Two packages may be specified to provide for monaural and stereo.

At this point, we observe that there may be a good deal of difficulty in transporting tools across APSEs, even if we have rigorously defined KAPSE interfaces. The issue here is ultimately that of hardware functionality. For example, a screen editor can hardly be transported to an implementation consisting of a teletypewriter. If it is assumed that "all" APSEs to be built will only utilize teletypewriters, there may be a temptation (among KAPSE interface standard specifiers) to ignore the interface requirements for non-teletypewriter input-output devices. Even though this example is somewhat unrealistic, it shows that a danger exists if a short term KAPSE interface standard (reflecting present APSE tools and input-output devices) becomes a "de facto" long term standard. Consequently, the guiding philosophy should not be to design KAPSE interfaces to bias future tool builders. This may happen if certain short-term KAPSE interface efforts are to be regarded as long-term standards. This effort may not standardize KAPSE interfaces that are general (or "extensible" — see [8]) enough to allow future tool builders any leeway. Consequently, these future tool builders would be forced to bypass the KAPSE and build a tool that interfaces directly with the host operating system. This tool would be easily transportable (even though

it would have been within the technology to insure that the tool be functionally transportable had the suitable KAPSE interfaces existed).

An analogy with the Ada language can serve as a possible paradigm. Just as not every Ada program utilizes every Ada language feature, we anticipate that not every APSE will utilize every (long term standard) KAPSE interface. Different tools will use different subsets of interfaces (just as different Ada applications programs rarely use every Ada language feature).

4. Models for KAPSE Interfaces

KAPSE interfaces that support, as Stoneman says, "all devices of these types," should conform to a model. This model must be specified so that conformance can be checked, and we must also be careful so that the model does not restrict any innovation in device technology.

Two good prototypes for these KAPSE interfaces have appeared in the literature. They both define their device models somewhat differently.

The first set of prototype KAPSE interfaces is found in a collection of Modula-II modules [5]. The input-output support of interactive devices is defined in terms of the specified interfaces: the "virtual devices" are those entities that conform to the Modula-II modules. These "packages" contain some informal semantics associated with the device functionality discussed above. Some examples of the semantics of these devices are contained in the Modula-II "packages" called LineDrawing, Mouse, and WindowHandler.

A second prototype for these KAPSE interfaces is an international graphics standard called GKS (Graphical Kernel System) [6]. GKS defines input and output in terms of a set of "logical" devices. Interfaces to these devices may be implemented in a number of ways. For example, the abstract devices for input can be specified in Ada by

```
type DEVICE is (CHOICE, LOCATOR, PICK, STRING, VALUATOR);
```

The literal CHOICE can model the buttons ("eyes") on a mouse, LOCATOR models position (of a moving mouse), PICK can be used for menu selection, and STRING and VALUATOR provide for the input of text string and the input of a real number, respectively. GKS also defines certain "attributes." These are classified according to "primitive" (pattern types and indices), "segment" (visibility, highlighting, segment transformations), and "workstation" (pattern areas, representations).

The GKS approach is to first define the properties of an "abstract," generic device and then worry about implementation of the interfaces. Any programming language would be suitable for this implementation. Recent articles speak of the suitability of BASIC and PASCAL for implementing GKS. The Modula-II approach is to first define the interfaces (in terms of Modula-II modules) and then worry about finding devices that can conform to these interfaces. The difference in these approaches are that the GKS approach seems more "top-down," and thus more suitable for long range specifications. The Modula-II approach seems more "bottom-up," and therefore more suitable for a short-term effort.

The need to concentrate on long term standards should neither be underestimated, nor should one dismiss these newer programming environments as being several years away. One interesting project involving a "newer" programming support environment is being sponsored by the Science and Engineering Research Council(SERC) in the United Kingdom[7]. This project is developing a high-quality, "commercially viable" personal workstation, including high resolution graphics, and also having the ability to input graphical and non-graphical input (especially voice). As far as standards are concerned, SERC has picked GKS for graphics, X.25 for networks, and Berkeley Unix for its "KAPSE." Ada is scheduled for SERC implementation. (The non-Ada SERC environment was targeted for completion in June, 1982.) If we want to exploit these tools from these newer environments by transporting them to other APSEs, we should insure that a long-term KAPSE interface standard be developed that will support the device functionality that these tools will use.

5. References

- [1] Gargaro, A., "Program Invocation and Control," KITIA NOTE WG.1-A002.3, October, 1982.
- [2] Cox, F., "KAPSE Support for Program/Terminal Interaction," KITIA NOTE WG.1, November, 1982.
- [3] Lindquist, T.E., "A Taxonomy of Command Language Features and Needed Underlying Support," KITIA NOTE WG.1, October, 1982.
- [4] Morse, H.R., ARPANET MESSAGE TO KITIA WG.1, 14 MARCH 1983.
- [5] Wirth, N., PROGRAMMING IN MODULA-2, Springer-Verlag, New York 1982.
- [6] Bono, F.R., et al, "GKS-The First Graphics Standard," IEEE COMPUTER GRAPHICS, VOL.2, NO.5, July, 1982.
- [7] Hopgood, P.R., Witty, R.W., "PERQ and Advanced Raster Graphics Workstations," IEEE COMPUTER GRAPHICS, September, 1982.
- [8] Standish, T., "A Philosophy for a Tool Extension Paradigm," KITIA DRAFT POSITION PAPER, August, 1982.

EVOLUTION OF AN APSE INTERFACE MODEL

Timothy Lyons
Software Sciences Ltd. United Kingdom

Introduction

This working paper traces the evolution of a model for the interfaces present in an Apse. The various models that were developed are explained, and their deficiencies are outlined (rather than just explaining the current model) as a means of giving greater insight into the models.

The models are based on the current Apse designs (that is the US Army ALS, the US Air Force AIE, the UK Ada Study and Olivetti's PAPS). However, it must be emphasised that misunderstandings are solely the responsibility of the author.

The four designs mentioned are sufficiently similar in their overall architecture that the models, with the exception of minor details, apply to all four designs, but of course with different terminology for each. We feel it is most important to gain an understanding of the architectures and interface models of the existing designs, as a starting point, rather than to try to develop an abstract or "ideal" model.

Why Have an Interface Model

It could be argued that a model of the interface is unnecessary, and that all the interfaces could simply be listed one by one. A model however, allows classification of the interfaces into different types, with the potential for different descriptive methods for each type. In addition it might provide a means for checking whether the list of interfaces is complete.

MODEL 1 - Overall System Architecture

We have been using an overall architectural model to explain the relationships between components, for some time. A simple version of this, which captures the essential points is shown in figure 1, while figure 2 shows a more detailed version including the target.

If we consider just the host, what we (and the other development teams) have called the Kapse interface (shown on the diagram as the public Kapse interface) is provided by the syntax and semantics of a number of Ada packages which are link edited with, or in the same address space as, the user's Ada code. The run time system for the host is similarly linked with the Ada code.

A certain amount of processing or buffering may be carried out by what we call the Kapse packages (eg AIE Kapse I/F packages) but most of the processing is carried out by what we call the Central Kapse or the Kapse database.

All the developers describe the Kapse interface in terms of the facilities offered by the Kapse packages, rather than those directly offered by the Central Kapse.

The developments differ in the following minor respects.

- a) exactly which packages constitute the Kapse interface. For example figure 2 (drawn for 1980 Ada) shows TEXT_IO as a Kapse interface, but other developers may regard this as a higher level interface built on a BASIC_IO interface, with the latter being the Kapse interface.

- b) the exact mechanism of communication between the Kapse packages, the run time system and the Central Kapse. We show all communication as passing through the run time system, but alternative architectural details are possible, for example with Kapse packages communicating directly with the Central Kapse as well as with the run time system.
- c) we have shown the Kapse database as a separate item from the Central kapse. It may instead be regarded as an integral component of the Central Kapse.

This interface model provides an effective description of the implementation of the lowest levels of interface in the system. However it does not deal with other interfaces, for example between the system and the user.

MODEL 2 - Single Level Model

This model is based on that developed by C. Forrest of TRW in support of the KIT. Figure 3 shows the (public) interfaces in the UK Ada study design drawn according to this model.

The model attempts to show all the interfaces in a single uniform level.

Interface K is the public interface to the Kapse, that is, the same interface as in the previous model. Now, for clarity, the internal implementation details of Kapse packages and central Kapse etc are not shown. These internal details are in any case irrelevant to the user.

U1 is the user interface directly to the host operating system. Generally, the aim is that the host should be hidden as far as possible, but this interface may perhaps be needed for logon. U2 is the user interface directly to the Kapse, for example for break in or emergency abort of programs etc. I1 is the special interface by which the Kapse invokes (activates) the user's command language interpreter (normally the standard Kapse CLI) when the user logs on. U3 is the user interface to the CLI,

consisting of the syntax and semantics of the user commands. I2 to I9 are the various invocation interfaces to the main tools, by which any initial start-up parameters are passed; similarly U4 to U11 are the various user interfaces to the tools. There are certain tools, to do with maintenance of the system, which may be invoked and used in a special way, these are shown with interfaces I10 and U12. Finally, the inter tool compilation interface covering the structure and use of the program library and intermediate languages is shown as M.

The difficulty with this model lies in its attempt to represent things which are at different levels at a single level. For example, consider the interface U3 between the user and the command language interpreter. This is entirely a conceptual interface; the CL1 does not really communicate directly with the user, but does so by I/O calls to the Kapse. The Kapse in turn passes these requests to the host operating system, and thence to the terminal. The diagram shows both levels of interface together.

MODEL 3 - Two Level Diagram

In an attempt to remedy what was seen as the main difficulty of the previous model, a two level model was developed, while retaining the previous overall structure of a diagram showing interfaces between components.

The two levels are termed the "mechanistic" level and the "conceptual" level. This layering is suggested by that of the Open Systems Interconnection Reference Model (OSI) of the International Organisation for Standardisation (ISO). However, the contents of the layers is developed entirely independently of the OSI model. The lower mechanistic layer provides the facilities on which the upper conceptual layer is built. The terminology is quite different from that of the ISO reference model.

The mechanistic layer shown in figure 4 describes how components of the system are joined together at the lowest level, and the lowest level

facilities for communicating between them. In general, this level is not concerned with information content but with the transfer of concrete items across interfaces.

Interface K is again the same public interface to the Kapse. Again, as in the previous models, the Ada code interfaces to the run time system, but this interface is not one which is explicitly seen or is identifiable to the Ada programmer. The RTS provides certain facilities defined in the Ada LRM, which it is not convenient to embody inline within compiled Ada code. This will typically include code to implement the Ada tasking mechanism, and possibly some aspects of storage allocation etc. The RTS may be target dependent, in the same way as machine instructions are target dependent. This target dependency is of no concern to the programmer, since he has no direct access to the interface, rather it is the responsibility of the compiler to generate the necessary calls on the interface. The facilities offered by the RTS are no more and no less than those required to support the appropriate parts of the LRM.

Interface S is to the common support packages, such as the Diana manager, program library manager, symbol table manager etc. H1, H2 and H3 are various interfaces to the host or host operating system.

On the target, we consider only the implicit interface between the Ada code written for the target and its run time system.

At the conceptual level, we are concerned with the semantic or information content which is passed across interfaces between components. At this level we are not concerned with the fact that, for example, the command language interpreter uses a Kapse primitive to read commands from the user, but regard this as a direct interface between the CLI and the user.

It should be recognised that all the interfaces at this level are realised by some combination of one or more of the mechanistic interfaces described above. However, in common with other interfaces, the method

of implementation is irrelevant, and may in fact vary without affecting the interface. Thus, this level defines expected information content, and the way in which data is moved around the system is not relevant.

As a specific example, the conceptual level user interface to the CLI specifies the syntax and semantics of the CLI command language. This is implemented at the mechanistic level by sequences of bytes. The conceptual level interface specifies the meaning of valid sequences of bytes. As another example, a particular tool will make certain assumptions about the structure and meaning of a Diana tree; for example that it contains only valid nodes (that it is well formed) and perhaps that it corresponds in some way to a valid Ada program. Certain of these checks are carried out by the packages that provide the Diana abstract data type, and these are described at the mechanistic level. Other assumptions which are not provided at the mechanistic level are described at this conceptual level.

In many cases, the interface at the conceptual level is described by a grammar and its associated semantics. The grammar defines the legal sequences of terminal symbols, that is, the sentences of the language. The mechanistic level is generally responsible for the transport of individual terminal symbols, irrespective of the meaning of the complete sentence. Viewed in this way, the system, at this level, consists of a number of active components which generate or act on sentences according to their defined semantics, and one passive components which provides storage and retrieval of sentences. This will normally be the Kapse database, but cases where sentences are passed between tools, or from one place to another within a tool do not logically differ.

The types of interface at this level are shown in figure 5.

Interfaces U1, U2, U3 and I1 are exactly as in the previous model. Interfaces U4 represent the tool invocation interfaces, and are shown as interfaces directly with the user. Interfaces U3 represent the user command interfaces to the different tools. Interface C represents the

information which the debug tool needs to know in order to make effective use of the monitored context interface. It may include details of code generation algorithms, standard uses of registers, layout and allocation of store, in particular stacks, layout and use of internal data structures, in particular those for tasking and exceptions etc. This interface will be entirely target dependent, in just the same way as machine instructions are target dependent.

Interface D represents the various program data interfaces. There are many such interfaces, which all represent the meaning of data passed from other programs, typically via the database. For the CLI, this represents the syntax and semantics of the command language. For the debug tool, this represents the meanings of the various pieces of symbol table and other program maps etc. which are read using the various mechanistic interfaces. This interface also describes such things as the structure of the compiler listing output, which at the mechanistic level is simply a stream of bytes.

Two minor points should be made when comparing the diagram for this model with that for the previous model. First, there is no particular significance in the grouping of the tool invocation and tool command interfaces into two large sets, namely U3 and U4 in this model, rather than a series of interfaces as in the previous model. Such grouping is rather arbitrary, and either approach fits equally with the overall model. Secondly, this model suggests that the tool invocation interfaces are seen as interfaces with the user, rather than with the CLI. Again within the limits of the model, the choice is rather arbitrary and of no particular significance; however this point does relate to the defect in the model.

There are a number of difficulties with this whole approach, which are clarified by examination of this model.

Although the sentence store does provide a neat model for the way in which, for example the CLI, accepts the syntax and semantics of valid sentences within its own particular language, the sentence store itself

seems rather artificial. Moreover, output from one tool, for example the editor, may be in a different language (ie just a string of characters) to the same database object when it is used as input, for example to the CLI; the sentence store therefore appears to carry out language conversions.

Another problem area is that, depending on the method of invocation, a tool might take its input direct from an interactive user, or from a database object. Although some tools may behave differently in these two cases, consider the simple case where the tool does not care where the input is from. The interface is then the same for both user and sentence store, and representation on the diagram is at best awkward. Similarly, we have already mentioned the difficulty in deciding whether the invocation interface which a tool offers is with the user - he certainly needs to know what parameters to type - or with the CLI, which must pass the parameters to the tool. There is a spectrum from a dumb CLI which just passes on the parameters given by the user, to a highly intelligent one which knows about certain tools and the parameters they require, and constructs these parameters without explicit user direction. Irrespective of the position on the spectrum, the tool itself expects exactly the same thing, and yet these appear to require different representations in this model.

MODEL 4 - Multi Layer Interface Lists

The problems which arise with the previous interface models are due to a fundamental flaw in the whole approach. The models are based on the assumption that interfaces are a link or communication between two components, for example between the user and the CLI. This view is intuitively attractive, since the user is clearly interacting with the CLI and there is an obvious need to describe the rules of this interaction. Furthermore this view allows a diagrammatic representation and this can be seen as an aid to checking whether all relevant interactions/interfaces have been considered. Finally, this view does seem appropriate to the lower level interfaces, as is seen by the consistency of these interfaces over the various models.

Despite its attractions, however, it is clear that an interface is not a function of a link between two components, but is instead simply a property of a single component; that is it is a specification of what that component offers to the outside world, independently of who wishes to take up that offer. A diagrammatic model works well enough for the lowermost interfaces, since the components are really only plugged together in one way, but at other levels the arbitrary variety of connections defeat any such representation.

We therefore conclude that the only way to represent the interfaces at a level such as the conceptual level already discussed, is simply to list each one. As an example, the CLI has an invocation interface; this is normally used during login to start up the user's CLI, but could also be used directly by the user or even by another tool to start up another instance of the CLI. The CLI also has a command language interface; this may be used interactively by the user or by submission of a command file.

In order better to model the interfaces in the existing Apse developments and to provide a basis for comparison, we extend the model in the previous section to four levels:

- Level 4 Conceptual
- Level 3 Common service routines
- Level 2 Mechanistic
- Level 1 Physical.

Level 1 contains interfaces at the level of read/write a disc block at a physical disc address. This level is included since it is the level at which the existing Apse developments, specifically the ALS and AIE, are entirely compatible. In the case of the AIE, the UK Ada study and the Olivetti PAPS, this is probably about the level at which the Central Kapse communicates with the host operating system; that is it is used to implement the Kapse. In the case of the ALS, the Kapse uses the host's filing system, and so this level exists somewhere within the host operating system, perhaps at the boundary between the file handler and the device driver. Explicit recognition of this level is perhaps rather more for didactic rather than practical reasons.

Level 2 represents interfaces at about the level of standard Ada Input/output, or BASIC_IO and AUX_IO of the ALS. This is the level that is shown as K in the earlier diagrams.

Level 3 represents the common services such as symbol table, intermediate language and program library handling. These are separated out from the level 2 interfaces since they are thought of quite differently by the Apse developers.

Level 4 represents the same high level interfaces as in the previous model, such as the CLI command language etc.

Note that although the overall architecture of the current Apse developments is well represented by Model 1, this four level model only considers the package interface at level 2. We do not consider in this model whether the interface between the linked program and Central Kapse, or the Central Kapse and host or Kapse database, should be considered as further interfaces at level 2, or as a lower level. Such extensions to the model could be considered, given a clear understanding of this model and of the purpose of such an extension.

It must be clearly recognised that the choice of layers, and the allocation of functions and facilities to layers is fundamentally arbitrary, and a matter for agreement, given guidelines, rather than a process susceptible to logical deductions from fixed rules. This is clear in the ISO OSI model, and is equally true here. In particular there has been a strong desire to "define" what should appear at level 2, so that a determination could be made for a particular Apse development whether a particular facility should or should not be in level 2. (Note that we have taken exactly the opposite approach here, of determining what is present in each design, and developing a model to describe these designs). In fact, such a definition is not possible, since an infinite number of slight shifts in functionality can be made within any overall framework. Therefore the only descriptive mechanism is simply to list interfaces which are deemed to be at each level, or at least to list representative examples and allow extrapolation.

Given the model of interfaces, there are a number of questions relevant to standardisation.

- a) what should the levels be called
- b) should levels be bypassable
- c) which levels should be standardised

Naming

We emphasise again that these levels are derived as a descriptive model of existing designs, rather than as an abstract view of what the structure of an Apse "should" be.

Stoneman, which coined the term Kapse makes it quite clear that the word is intended to describe both level 2 and level 3 interfaces.

On the other hand, for the current Apse developments, there is an almost universal use of the word Kapse to mean just the level 2 interfaces.

Recognition of this dichotomy of usage is clearly vital to a sensible discussion of interfaces in Apses. Given this recognition there are two possibilities.

- a) retain the original Stoneman definition of "Kapse" to mean both level 2 and level 3 interfaces.
- b) change the definition to mean only the level 2 interfaces.

There are arguments in favour of both possibilities which will not be pursued here.

Given this viewpoint, the statement that "the level 3 interfaces should be in the Kapse" can have two possible meanings.

- i) the statement is an argument for retention of the original Stoneman definition of Kapse. As such it is not an argument but simply an assertion. There are reasonable arguments for changing the definition (such as present common usage) and so better arguments than assertion are needed to retain the definition.
- ii) the statement may reflect some conception on the part of the speaker as to a qualitative difference between a "Kapse" interface and any other interface (a difference such as method of implementation or security or integrity). However, as has been made clear, each interface level is simply a list of facilities with associated syntax and semantics. If particular implementation or security or integrity constraints are required for certain level 3 interfaces, then these requirements are part of the specification of the semantics of that interface. Such semantics do not affect in any way the simple labelling issue as to whether the term Kapse should encompass level 3.

Bypassability

The present Apse designs do not generally intend tools or programs to access facilities at lower levels than level 2, except for ALS which allows access to host operating system facilities. It is not at all clear to what extent this denial of access is enforced. In general, there is some restriction because an interface very close to level 2 is implemented by a different machine mode or in a different address space.

The question then arises as to whether the facilities offered by level 3 should also be not bypassable. That is, whether objects such as symbol tables, which are normally accessed by packages defined at level 3, should not be accessible by level 2 facilities. Logically, if both level 2 and level 3 were not bypassable, then they would become one level of interface; certain files, for example symbol tables being accessed by what had been level 3 facilities, and other files, for example text files being accessed by what had been level 2 facilities. In fact, it is the architectural

structure of the present Apse designs which draws a distinction between level 2 and level 3. In the present designs, level 3 facilities are provided by ordinary Ada packages which are linked in with the user's program in the same way as any other Ada package. Level 2 in contrast is mainly provided by special requests to the Central Kapse.

To put the argument another way, given the lists of facilities offered by the current systems, certain files, for example text files, are only accessible by level 2 facilities; there are no relevant level 3 facilities. If level 3 were not bypassable, then either:

- a) the text access facilities would have to be brought up to level 3 or
- b) not bypassable means not bypassable for certain files but bypassable for others.

The feasibility of denying access to certain database objects except to certain authorised packages was considered during the UK Ada Study, where it was known as "package based access rights". Given the overall system architecture, and the desire to be able to host that architecture on a variety of host machine architectures, it was concluded that there were too many technical difficulties for the facility to be included in the Apse system design. Clearly, the requirements have to be considered carefully in the light of possible system architectures.

Standardization

Given the general intention that level 2 should be (or should be regarded as being) not bypassable, then given appropriate restraints on the way programs are written, it is not necessary to standardise lower than level 2.

Conversely, in order to ensure maximum portability of tools, it is necessary to standardise at all three levels, that is levels 2, 3 and 4.

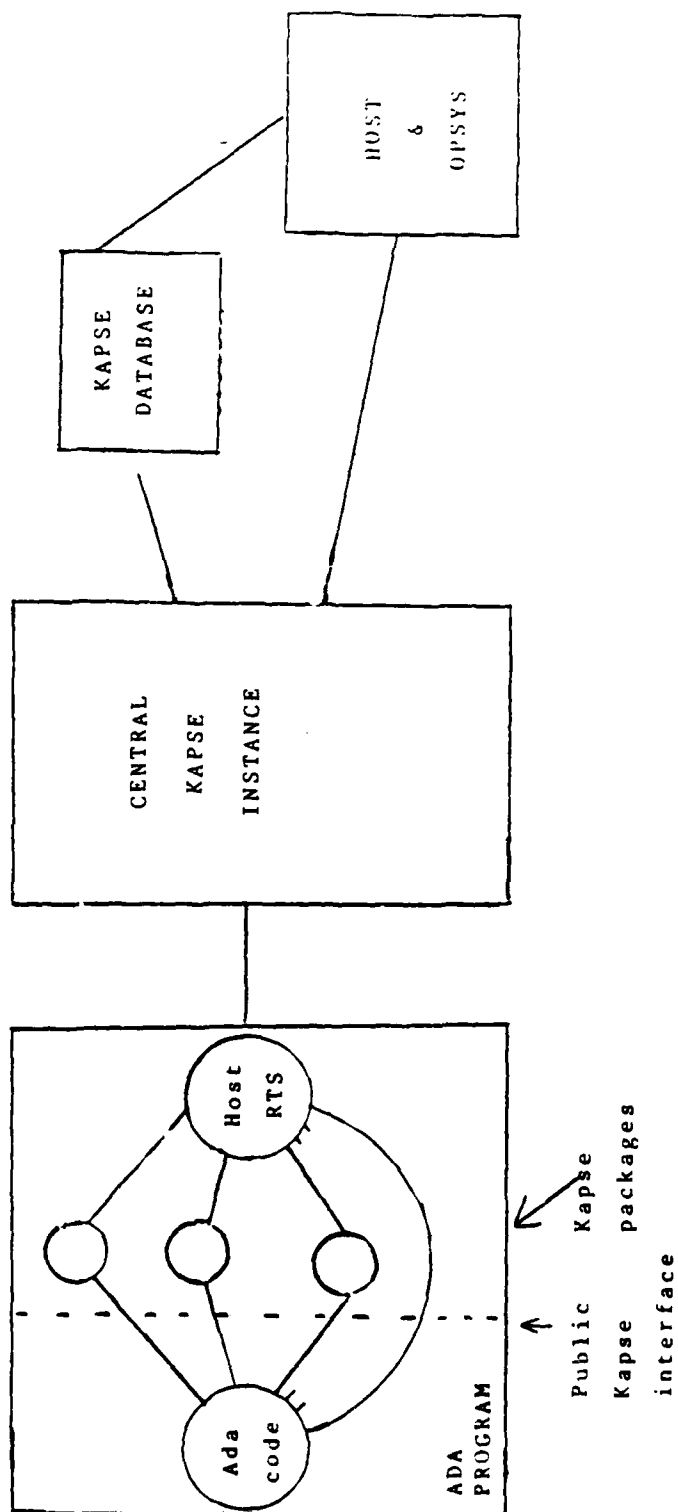


Figure 1 - Model 1 Interfaces (simplified)

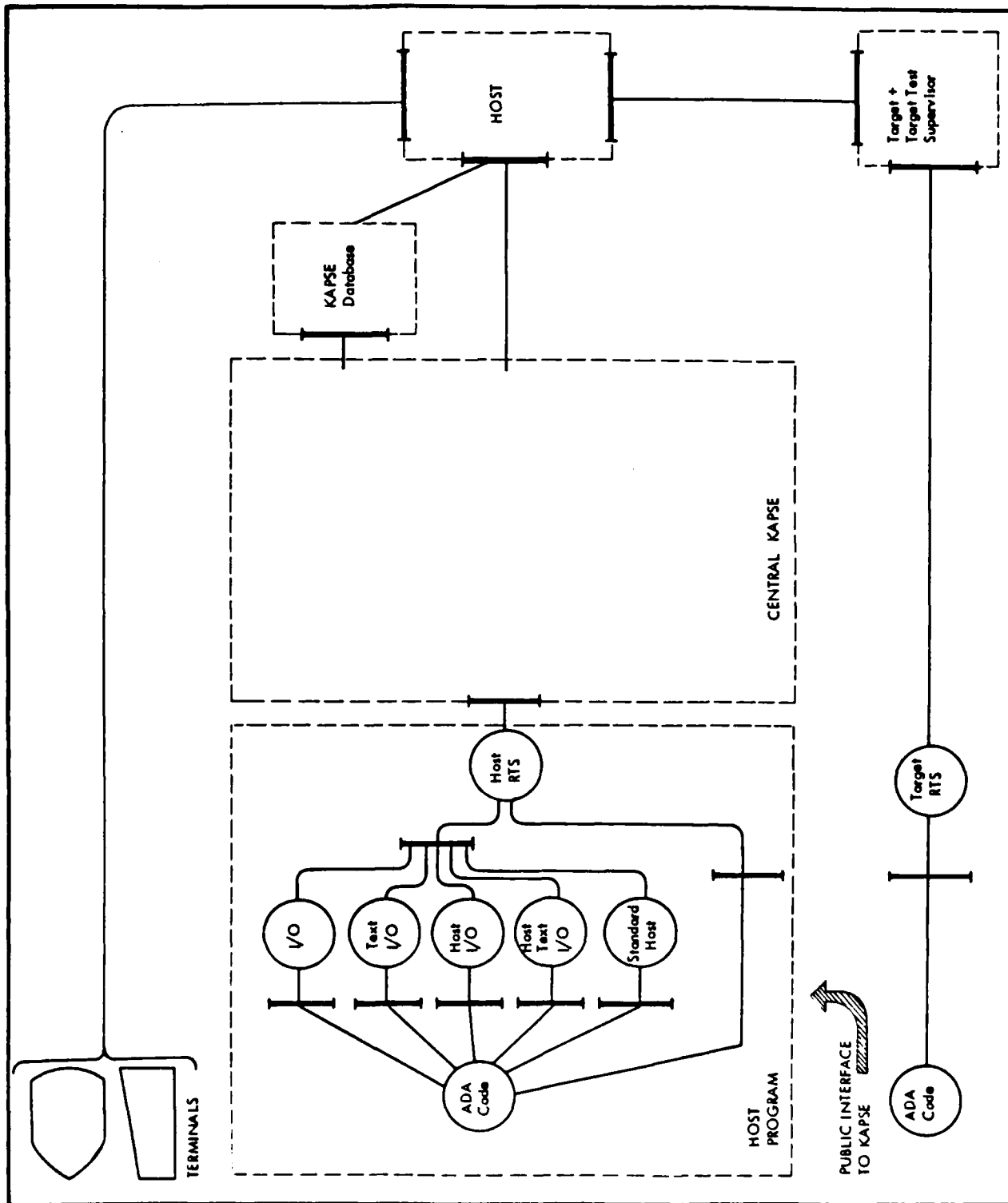
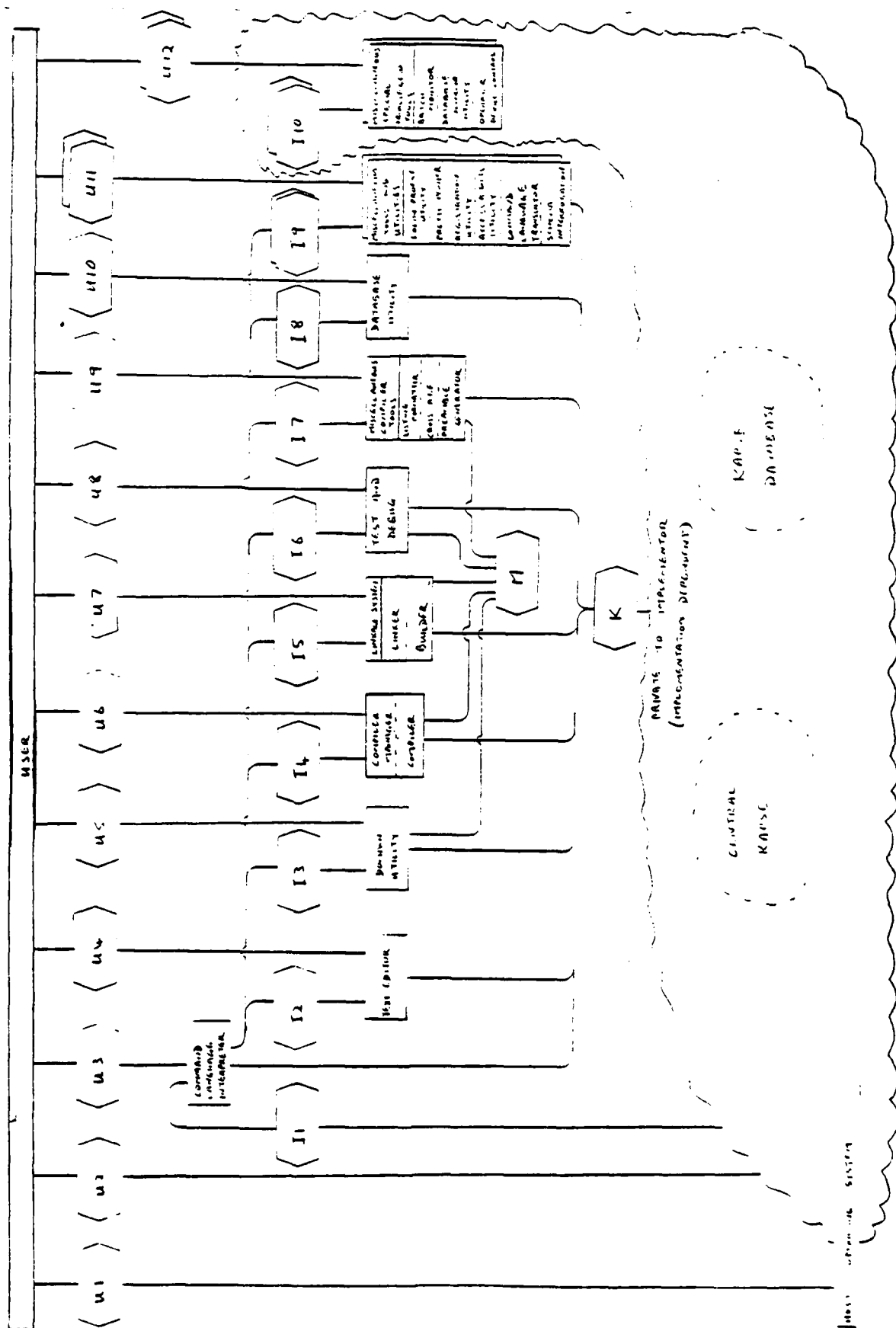


Figure 2 - Model 1 Interfaces

Figure 3 - Model 2 Interfaces



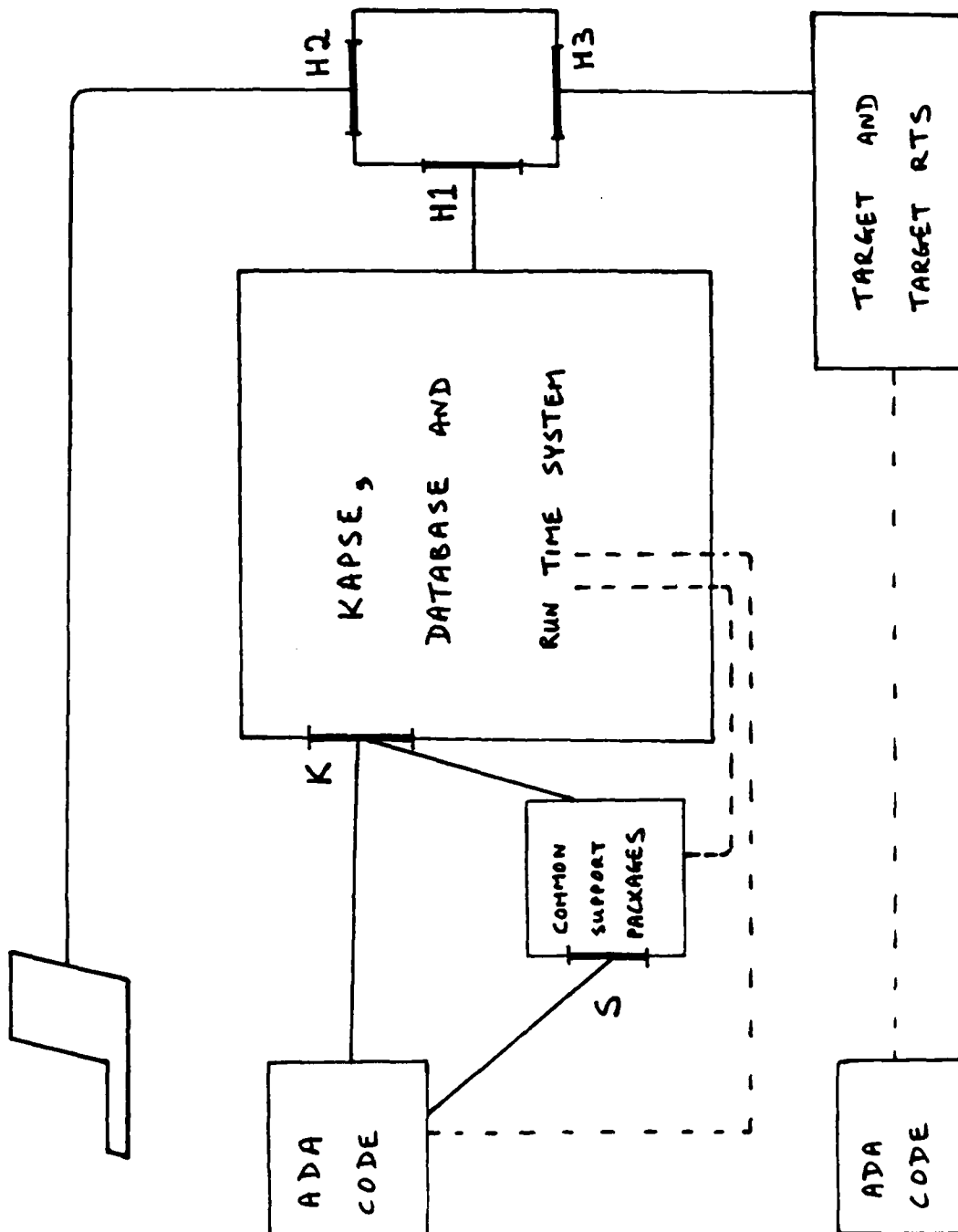


Figure 4 - Model 3 The Mechanistic Interfaces

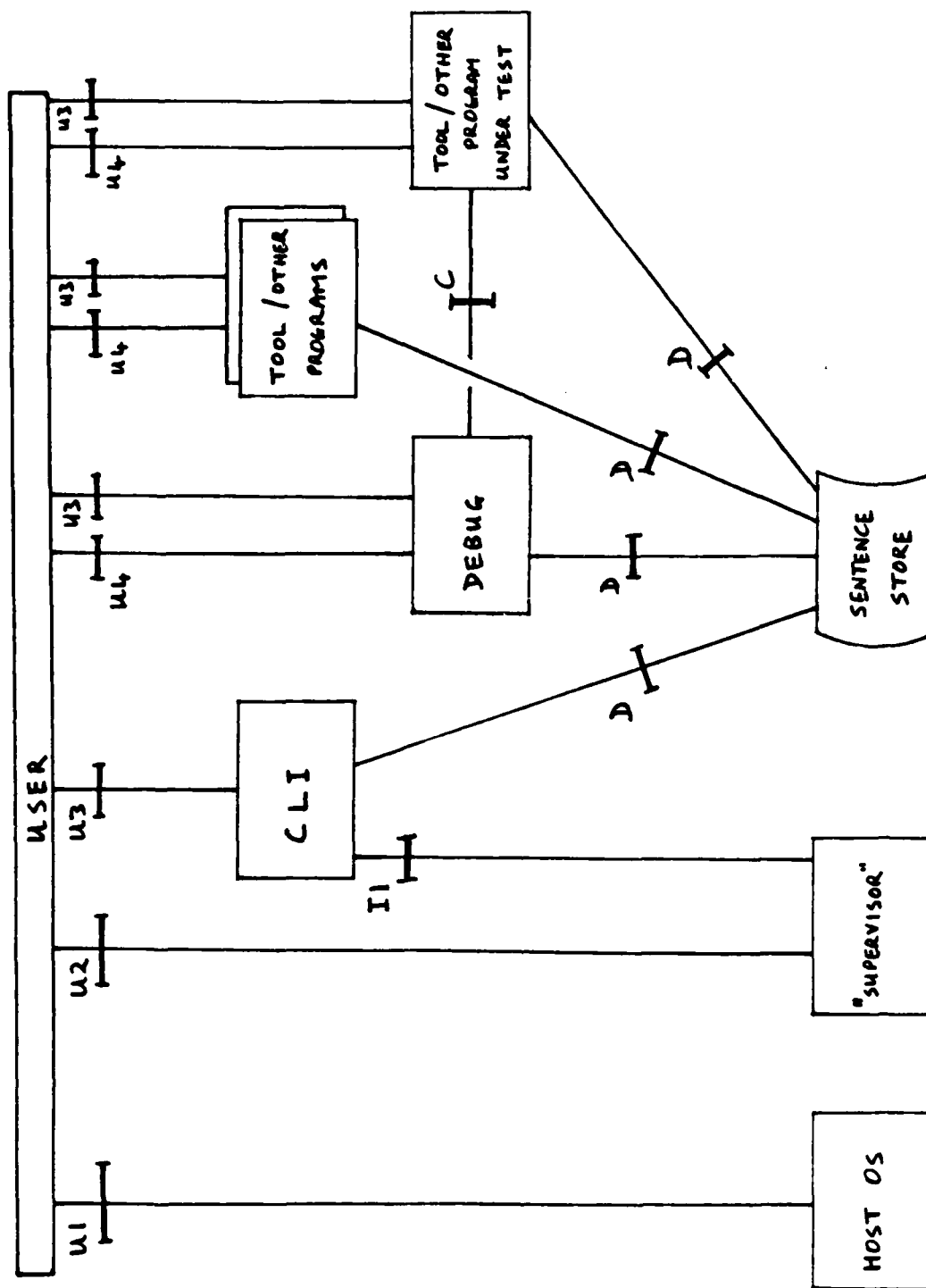


Figure 5 - Model 3 The Conceptual Interfaces

APPENDIX A

CONFIGURATION MANAGEMENT SYSTEM
INTERIM REPORT ON INTERFACE ANALYSIS

Prepared for
NAVAL OCEAN SYSTEMS CENTER
271 Catalina Blvd., Bldg. A-33
San Diego, California 92152

Under
Contract N00123-80-D-0364

27 AUGUST 1982

COMPUTER SCIENCES CORPORATION

6565 Arlington Boulevard
Falls Church, Virginia 22046

Major Offices and Facilities Throughout the World

TABLE OF CONTENTS

<u>SECTION 1 -- INTRODUCTION</u>	1-1
<u>SECTION 2 -- CONFIGURE'S INTERFACE REQUIREMENTS</u>	2-1
2.1 Command Language Interface	2-1
2.2 Program Invocation and Communication Interface	2-2
2.3 Database Services Interface	2-3
<u>SECTION 3 -- ADA LANGUAGE SYSTEM INTERFACES</u>	3-1
3.1 Command Language and Process Communication Interfaces	3-1
3.2 Database Interfaces	3-1
3.2.1 Database Structure	3-1
3.2.2 Database Services	3-3
<u>SECTION 4 -- ADA INTEGRATED ENVIRONMENT INTERFACES</u>	4-1
4.1 Command Language and Process Communication Interfaces	4-1
4.2 Database Interfaces	4-1
<u>SECTION 5 -- CONFIGURE'S PORTABILITY STRATEGY</u>	5-1
5.1 Environment Independent Processing	5-1
5.2 Environment Dependent Processing	5-4
5.3 Interface Correspondence	5-5
5.3.1 Database Services	5-5
5.3.2 Standard and Message Output	5-10
5.3.3 Tool Invocation and Communication Services	5-11
<u>APPENDIX A - REFERENCES</u>	A-1

SECTION 1 -- INTRODUCTION

The purpose of the Configuration Management System (CMS) is to automate many of the configuring activities involved in software development, testing, and maintenance. CMS consists of an Ada Programming Support Environment (APSE) tool, Configure, that is being designed to be portable and to be hosted on both the U.S. ALS and the U.S. Air Force's AIE.

This report is an interim analysis of the interface requirements of Configure, along with the relevant interfaces of the Ada Language System (ALS) and the Ada Integrated Environment (AIE). It must be recognized that at the present time neither the ALS nor the AIE exists, and that the available documentation for both systems is incomplete and presently under revision. Moreover, although a Program Performance Specification for Configure is now available, the detailed design is incomplete. Therefore the present report is a preliminary analysis only, with the intent of identifying the aspects of the ALS and AIE that may be expected to impact the design and transportability of Configure. The report concludes with a summary of the design decisions that have been made to accomodate the ALS and AIE interfaces.

The reader should be familiar with Stoneman [12], the CMS Program Performance Specification [7], the ALS System Specification [5], and the AIE Interim Technical Report [2].

SECTION 2 - CONFIGURE'S INTERFACE REQUIREMENTS

The Configure tool that comprises CMS has three classes of interfaces that must be dealt with to ensure transportability: command language, program invocation and communication services, and database services. This section will discuss Configure's requirements in each of these areas.

2.1 COMMAND LANGUAGE INTERFACE

The configuration objects processed by Configure contain Command Language (CL) statements that must eventually be processed by a Command Language Interpreter (CLI) on the hosting MAPSE. Configure performs minimal but important pre-processing on these statements, and needs to be able to identify database object names, Configure macro definitions, and inter-statement delimiters.

In pre-processing object names, Configure may need to append revision indices. Configure is otherwise indifferent to object name syntax and semantics, and does not need to deal with directory, partition, or variant structure.

Configure allows configuration objects to include macro definitions that are to be expanded in both the dependency rules and in the command lists. For the latter expansion, Configure must be able to recognize the presence of a macro name in a CL statement. It is recognized that there is no universally transportable syntax to ensure this recognition, and the present definition, specifying "#identifier", is a prototype syntax only. In order for macros to be useful and convenient, it is necessary for the hosting syntax to permit the specification of an equivalent macro name syntax that does not conflict (in a syntactic recognition sense) with other CL structures.

2.2 PROGRAM INVOCATION AND COMMUNICATION INTERFACES

After pre-processing CL statements, Configure transmits them to the MAPSE's CLI for interpretation. There are three techniques that Configure could use for this transmission:

- (1) invoke the CLI once per CL statement;
- (2) invoke the CLI to process an entire group of statements;
- (3) establish a "pipe" to the CLI over which command lines (not necessarily statements) could be sent one at a time.

Each of these techniques has advantages and disadvantages.

The first method, invoking the CLI once per statement, obviously incurs a significant recurring overhead for process invocation. It requires that Configure be able to recognize precisely what constitutes a CL statement, including compound statements. Since each statement is processed by a separate CLI invocation, it is difficult to pass context (e.g., CL variable values) within which subsequent statements are to be interpreted. On the other hand, Configure retains some visibility into the "granularity" of statement processing. This granularity facilitates better error messages; with this technique Configure can identify for the user precisely which statement failed and (possibly) why.

The second method, invoking the CLI once per group of statements, reduces the recurring overhead and, more importantly, allows a group of statements to be processed within the same context. Granularity is, however, reduced, and depending on the kind of error messages generated by the CLI, Configure may not be able to identify for the user the particular statement that caused an error. Further, it is not even guaranteed that a CLI will terminate upon encountering an error in a group of statements. Such termination can be considered to be a special feature, since

the normal mode for a CLI in interactive use is to report the error and then proceed to process the next statement. It must be noted that the first method, if compound statements are allowed, will be susceptible to a similar loss of granularity within the compound statement. •

The third method, that of establishing a pipe between Configure and CLI, is the best from the point of view of error reporting and control, while allowing context to be preserved for a group of statements. Actually two pipes are required, one to send CL statements from Configure to the CLI, and one for the CLI to report errors and confirmations back to Configure. This method would of course require an extremely cooperative CLI, as well as KAPSE support for pipes.

2.3 DATABASE SERVICES INTERFACE

For dealing with most aspects of the database, Configure's requirements are no different from other MAPSE tools. Configure's requirements to create, read, and write database objects, and to create, read, and write object attributes, are thus fairly prosaic. On the other hand, Configure has a very special interface with the history attributes.

It will be useful to reexamine the original Stoneman requirement for the history attribute:

Every object shall have a history attribute. The history attribute records the manner in which the object was produced and all information which was relevant in the production of the object. The history attributes shall contain sufficient information to provide a basis for comprehensive configuration control.

Since "configuration control" is not defined in Stoneman, this requirement is somewhat vague. However, a subsequent unpublished paper by Buxton [6] sheds some light:

A configuration is a well-formed group in the sense that structural rules, by which configurations are assembled as groupings of objects which may in some cases be shared with

one or more other configurations, are known to the system and so the possibility exists of automatic manipulation of configurations by the system. Thus, for example, in some methodological approaches configurations may be subject to automatic re-derivation; if one constituent object is changed it is possible for the system to determine and carry out all necessary consequential changes to all configurations making use of that object, however indirectly.

The key question is: what is the history attribute to be used for? There are three overlapping answers. First, the history attribute can be viewed as a "commentary" on the derivation of the associated object. Second, the history attribute can be used, as indicated above, to direct automatic re-derivations in the face of consequential changes. Third, the history attribute can be used to reconstruct a previous version of an object whose information content has been deleted.

The original concept for Configure was based on the realization that if history were maintained to a sufficient precision to satisfy automatic re-derivation, then reconstruction could be provided with minimal additional effort. In fact, the algorithms for re-derivation and reconstruction are nearly identical. Configure was originally conceived to be closely integrated with the history maintenance system proposed by Computer Sciences Corporation as part of the database for the Ada Integrated Environment. Under this design, Configure relieved the KAPSE of much of the responsibility for maintaining the derivation information. The notion of a configuration object not only includes identification of all the objects in the configuration, but also includes the command language scripts whereby these objects are derived from one another. This design obviated the necessity of having the KAPSE maintain derivation scripts.

In keeping with the above design, Configure requires a database to include in its history attributes only dependencies and date-time stamps. The dependencies, or dependency attributes, indicate the specific objects (or revisions thereof) that were

used in the immediate derivation of the associated object. Included in the dependency attribute must be a reference to the configuration object which provides the command scripts that were used in the derivation. The form of the dependency attribute is of minor importance to the functionality of Configure, but is significant when it comes to moving objects around a logical database or between databases. Configure only needs to extract logical object names from the dependency attribute.

Unfortunately, there are cases for which the history attributes of any "reasonable" database will be insufficient to provide a guaranteed basis for re-derivation or reconstruction. The problem is that a tool can access global data (other than file contents) within the system and use this data in creating files. If this global data is subsequently modified, the created files are not reconstructable. This dependence can be very subtle -- a tool can, for example, monitor system performance characteristics and use these in file creation. This is a pervasive problem; it is not feasible to preserve in the history attribute the totality of available global data, and there is no useful way of detecting and preventing a tool's access to such global data (this problem is strikingly similar to that of preventing "covert channels" that may violate computer security). In practice, this issue is unlikely to impair the utility of a tool such as Configure, although it must be recognized that reconstructability cannot be absolutely guaranteed. It should be fairly easy to establish a set of reasonable, though non-testable, guidelines that, if followed, will ensure reconstructability.

SECTION 3 - ADA LANGUAGE SYSTEM INTERFACES

The ALS interfaces are reasonably straightforward and well-documented. Except as indicated below, Configure can be hosted on the ALS with little difficulty.

3.1 COMMAND LANGUAGE AND PROCESS COMMUNICATION INTERFACES

A potential problem with Configure's use of the ALS Command Language Processor (CLP) is the lack of pipes, which constrains the form of the interaction (as discussed in Section 2.1). Fortunately, it is possible to request the CLP to terminate on encountering an error in a group of commands, and the CLP's output parameters will contain the statement index of the error-causing command.

3.2 DATABASE INTERFACES

The ALS database follows the model of a conventional hierarchial file system. The most significant extension is the provision for "variations" at any node in the hierarchy. Except for one major problem and a number of relatively minor details, the ALS database philosophy is fully compatible with Configure.

3.2.1 Database Structure

Below is a list of those aspects of the ALS database structure that impact the rehostability (transportability) or functionality of Configure.

(a) The ALS allows the most recent revision of a file to be modified, while Configure always adds a new revision if a file needs to be updated. Since only non-current revisions are automatically frozen, Configure must explicitly freeze the current revision to prevent non-Configure access from altering it. (Also, it must be noted, allowing modification of the most recent revision would violate Stoneman.)

(b) Configure would, in principle, allow a user to delete non-current revisions if nothing depended on them. The ALS does not allow such deletion, possibly resulting in a pervasive retention of useless intermediate revisions.

(c) The "variation" concept of the ALS is orthogonal to Configure. Configure will semantically ignore the variation portions of an object name, but will have to recognize the syntax. Since Configure will pass such names, unchanged except for the possible addition of a revision index, the Command Language Interpreter will be responsible for the semantics, including that for default variations, independently of Configure. If Configure were to recognize aliasing of file names (a feature that could possibly be added), it would have to understand variations as well.

(d) The ALS derivation attribute for a node is defined to contain "the name of the tool that created the node along with all parameters to the tool and the names of all other nodes used to produce the new node." The precise information that is written into the derivation attribute when a node is closed is in fact chosen by the tool that is closing the node.

This design has advantages and disadvantages. If the derivation attribute were the sole responsibility of the KAPSE, there would be no way for the KAPSE to distinguish between temporary (or unimportant) nodes and those nodes actually essential to the derivation. Frozen temporary files would then accumulate. On the other hand, every tool must now contain code to manage history attributes.

There is one aspect of the ALS's derivation attribute that may make it unreliable from the point of view of re-derivation or reconstruction. First, a derivation of a file may well entail the efforts of two or more processes. For example, process A may open a file X, call on process B to generate some intermediate results,

and then close X. It would be desirable to include in the derivation of X all (non-temporary) files opened by B. This inclusion is not performed by the ALS; one way to incorporate it would be to allow open files to be passed between processes.

3.2.2 Database Services

Consistent with the ALS database structure, the ALS database services do not otherwise impact Configure, except as noted below.

(a) One of the features of Configure is that it allows the reconstruction of the information content of an object based on its derivation history. The ALS does not, however, allow the information content of a frozen object to be deleted. Thus, one of Configure's major features becomes moot for the ALS. We recognize that the ALS was designed without a Configure-like capability, and thus was forced (in order to abide by Stoneman's precepts) to restrict deletions. An option now available to the ALS is to allow information content deletion if the content is reconstructable.

(b) Although the ALS documentation implies the availability of user-defined attributes, no functions are presently provided to create them. As indicated above, the system-defined derivation attribute is not entirely sufficient for Configure's reconstruction function, since it is not guaranteed to list all the objects actually used in a derivation and, in particular, it does not seem possible to ensure that the attribute will include the name of the relevant configuration object. One way of circumventing this problem would have been for Configure to define and manipulate an additional user-defined attribute.

SECTION 4 - ADA INTEGRATED ENVIRONMENT INTERFACES

From Configure's perspective, the AIE interfaces differ remarkably little from those of the ALS. However, given the lack of specificity in the currently available AIE documents, the following comments should be considered tentative.

4.1 COMMAND LANGUAGE AND PROCESS COMMUNICATION INTERFACES

There are only three significant differences in this area in the facilities provided by the ALS and AIE. First, the AIE Command Processor (CP) supports pipes, and thus in principle could support the more flexible and powerful form of Configure-CP interaction outlined in Section 2.2.

Second, the AIE CP will not terminate on encountering a command error, but will resume processing with the next statement. One hopes that this is an oversight, and that the different requirements for interactive and non-interactive invocations of the CP will be recognized.

Finally, there is no distinction between message (or error) output and standard output. It is clearly desirable to distinguish these, since many messages will be purely informational (as opposed to errors) and should not clutter the standard output which may be subsequently processed by another tool.

4.2 DATABASE INTERFACES

While the AIE database is conceptually far more complex than that provided by the ALS, this complexity has very little impact on Configure.

The most important interface with Configure is via the history attribute, and the AIE explicitly recognizes the two forms of the history attribute required for source and derived objects.

Essentially the AIE is placing revision maintenance under the umbrella of history, but the history attribute for a source object is simply an index used to extract a revision from a source archive, while for a derived object a program invocation script is provided. In addition, the script records "the parameters specified when the program was invoked, an array of copies of the history attributes of each object read as input, and a count of the number of objects created or modified as output." While the ALS makes it clear how the input objects are to be identified (it is the responsibility of the tool), this identification is not spelled out for the AIE. In particular, it cannot be determined how the AIE KAPSE distinguishes relevant files when one tool invokes a second tool to perform some intermediate processing.

The general ability of the AIE to support revisions is also unclear. For source archives there is no problem; all revisions are stored in a single file and they are indexed by "state". For derived objects, explicit support for multiple revisions is only discussed in the context of program libraries, where one component of the object name can be a "Virtual Memory Sub Domain (VMSD) number." There is apparently no automatic facility for selecting the latest revision of an object, or for incrementing the VMSD number when a new revision is created. In short, there appears to be no direct KAPSE support for revisions. Perhaps more support for revisions will be included within the VMM, otherwise a large burden will be placed upon the designers of MAPSE tools.

SECTION 5 - CONFIGURE'S PORTABILITY STRATEGY

As indicated in the Configure Program Performance Specification, Configure shall be designed to minimize the effort necessary to transport it to a designated APSE. In particular, Configure shall be designed to be transportable to both the ALS and AIE. While we have judged this transportability to be feasible, one general criticism can be made of both the ALS and the AIE, particularly with respect to the history management facilities in their databases. In both environments, these facilities are unique in that no MAPSE tools that the respective contractors are currently providing will directly interface with them. Since history management and configuration management are closely interrelated, one cannot easily be designed without the other. Thus we have found that in neither the ALS nor the AIE do the history management facilities accurately support the configuration management requirements.

To facilitate transportability, Configure will be divided into two disjoint sets of modules, the Environment Independent Part (EIP) and the Environment Dependent Part (EDP).

5.1 ENVIRONMENT INDEPENDENT PROCESSING

The EIP shall contain all modules required for internal processing except for those involving APSE-dependent service calls and responses. However, the EIP shall contain (in an APSE-independent manner) the service request specifications for the command language, database services, and program invocation and control services. From the point of view of the EIP, the EDP shall provide a simplified yet sufficient environment to implement these requests -- the EIP shall issue no direct requests to the hosting APSE.

Nevertheless, there are a number of design decisions for the EIP that are impacted by the choice of candidate APSEs. The most significant of these is the manner in which Configure communicates with the CLI. There is a clear semantic difference between the three modes of communication discussed in Section 2.1, and thus the choice of mode cannot be relegated to the EDP. In order to simplify transporting, Configure shall therefore invoke a single instance of the CLI to process an entire group of CL statements. It will then not be necessary for Configure to recognize individual statements, and a context can be established and maintained for statement interpretation.

The allocation of responsibility for maintaining the derivation attribute is a major issue from the point of view of designing a portable Configure, but is less important to the user-perceived functionality of the tool. The difficulty in having the KAPSE maintain this attribute automatically is that the KAPSE cannot distinguish between temporary (or unimportant) intermediate objects and those actually required for the derivation history. Moreover, a tool may indirectly access objects through the invocation of a second tool.

On the other hand, if Configure were to maintain the derivation attribute, it would include in it only the objects specifically referenced in the relevant dependency rule. In effect, this transfers the burden to the user, who must ensure that the dependency rule mentions precisely the correct objects. Unfortunately, except in simple cases, there can be no automated way of verifying the correctness of dependency rules. The chief advantage of this approach is that the KAPSE is relieved of an enormous burden, and derivation attributes are only maintained for those object under configuration control (i.e., whose derivation is specified by a configuration object).

There is some concern that the original Stoneman model has over-complicated the KAPSE, and that the inclusion in the KAPSE of a sophisticated database only marginally aids tool transportability while severely hindering KAPSE rehostability. A significant, though not pervasive, modification to Configure would be required for transporting to a future APSE with less KAPSE functionality.

The notion of "current working directory" must also be defined for Configure. Configure could use the current directory (or partition) of the process that called it, or it could use the partition in which the configuration object resides. To simplify transporting, we have chosen the first option, since it allows Configure to essentially ignore the problem. Thus all object names processed by Configure will be assumed to be relative to the current directory of the process. This decision is also relevant to the aliasing problem in general. For simplification, the present design for Configure will assume that distinct object names identify distinct objects, and Configure will make no attempt to verify that this assumption is in fact true. Such verification could be incorporated into a subsequent release.

As has been noted, Configure was designed to avoid the necessity of storing command language scripts with each revision of each derived object. These scripts, in a generic form, are instead stored in the configuration object, and a single instance of a script may suffice for numerous revisions of a single object. Since scripts may contain Configure macro names, it was tempting to allow these macros to be redefined on the command line invoking Configure. Unfortunately, such redefinitions would have to be stored with the consequent revisions, and this would be tantamount to storing the scripts in the first place. As a result, command line redefinition of Configure macros is not supported, the configuration object must be revised instead.

5.2 ENVIRONMENT DEPENDENT PROCESSING

The EDP is essentially an interfacing layer between the EIP and the hosting APSE. The EDP shall contain those modules required for the implementation of the EIP's service request specifications in terms of the services provided by the hosting APSE. While the specifications shall be the protocol with which the EIP shall request and receive responses of the APSE's services, the EDP shall be the translation of this protocol into the actual service requests of the APSE. In terms of Ada, the EDP shall contain the bodies of the subprograms specified in the above service request specifications.

The EDP shall be tailored to recognize, for the CL of the APSE CLI: command statements, database object names, and the command to invoke Configure. Also, the particular syntax used to distinguish Configure macro names may also have to be modified so as not to conflict with a given CL.

The EDP shall translate the database service requests of the EIP into actual APSE database service calls. It is recognized that a hosting APSE may not directly support the history attribute model needed by Configure. The actual history attributes provided may be a subset or a superset of those required. The transportability of Configure is highly dependent on the ease with which the Configure model can be mapped into that of an APSE. Further, the ability of the APSE to support user-defined attributes is crucial to transportability in many cases where the mapping is not exact.

5.3 INTERFACE CORRESPONDENCE

The subsections below indicate the correspondence between the interface requirements of Configure, as embodied in the Environment Dependent Part, and the relevant services provided by the ALS and AIE.

5.3.1 Database Services

1. Does a database object exist in the database?

EDP:

```
function OBJECT_EXISTS
    (NAME : in STRING) return BOOLEAN;
```

ALS:

Any KAPSE service that will return an error condition indicating that the database object name given to it does not refer to an existing object will suffice (e.g. READ_ATTR)

AIE:

Again, use of error return from a service call is sufficient (e.g. GET_ATTRIBUTE)

2. Is the information content of an existing database object present?

EDP:

```
function INFORMATION_PRESENT
    (NAME : in STRING) return BOOLEAN
```

ALS:

irrelevant (information content not separately deletable)

AIE:

Although it does not appear possible to delete only the information content of an object, the archive services for derived and source objects (particularly script retention) and the current status of a database object (i.e. on-line, archive-only) are relevant to this request. Their impact on the Environment Dependant portion of Configure is being explored.

3. Given a revision group that exists in the database, what is the identification of its most recent revision?

EDP:

```
function LATEST_REVISION  
  (NAME : in STRING) return STRING;
```

ALS:

```
procedure READ_ATTR  
  (NODE : in STRING_UTIL.VAR_STRING_RECORD;  
   ATTR : in KAPSE_DEFS.SHORT_IDENT_STRING;  
   START, LENGTH : in STRING_UTIL.STRING_INDEX_INT;  
   VALUE : in out STRING_UTIL.VAR_STRING_RECORD;  
   RESULT: out KAPSE_DEFS.I_O_RESULT_ENU);
```

AIE:

```
function GET_ATTRIBUTE  
  (NAME : in STRING;  
   ATT_LABEL : in STRING)  
  return STRING;
```

4. Open a Configuration Object for reading

EDP:

```
procedure OPEN  
  (FILE: in out OUT_FILE;  
   NAME : in STRING);
```

ALS:

```
procedure OPEN  
  (FILE: in out OUT_FILE;  
   NAME : in STRING);
```

AIE:

```
procedure OPEN  
  (FILE: in out OUT_FILE;  
   NAME : in STRING);
```

5. Read a Configuration Object that has been opened for reading

EDP:

```
function GET_LINE  
  (FILE : in IN_FILE) return STRING;
```

ALS:

```
function GET_LINE  
  (FILE : in IN_FILE) return STRING;
```

AIE:

```
function GET_LINE  
  (FILE : in IN_FILE) return STRING;
```

6. Close object that has been opened for reading

EDP:

```
procedure CLOSE  
  (FILE : in out IN_FILE);
```

ALS:

```
procedure CLOSE  
  (FILE : in out IN_FILE);
```

AIE:

```
procedure CLOSE  
  (FILE : in out IN_FILE);
```

7. Given a database object that exists in the database, read one of its history attributes

EDP:

```
procedure READ_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_NAME : in STRING;
   VALUE : out ATTRIBUTE_VALUE);
```

ALS:

```
procedure READ_ATTR
  (NODE : in STRING_UTIL.VAR_STRING_RECORD;
   ATTR : in KAPSE_DEFS.SHORT_IDENT_STRING;
   START, LENGTH : in STRING_UTIL.STRING_INDEX_INT;
   VALUE : in out STRING_UTIL.VAR_STRING_RECORD;
   RESULT: out KAPSE_DEFS.I_O_RESULT_ENU);
```

AIE:

```
function GET_HISTORY_REF
  (NAME : in STRING) return HISTORY_REF;
function GET_DIRECT_CONSTITUENTS
  (STATE : in HISTORY_REF) return HISTORY_REF_ARRAY;
function HISTORY_TIME
  (STATE : in HISTORY_REF) return CALENDAR.TIME;
```


8. Create a user-defined attribute for a given database object.

EDP:

```
procedure CREATE_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_NAME : in STRING;
   INITIAL_VALUE : in ATTRIBUTE_VALUE);
```

ALS: not clear that you can

AIE:

The fields for user-defined attributes are set when the parent COMPOSITE_OBJECT is created, using:

```
procedure CREATE_COMPOSITE
  (NAME : in STRING;
   COMPONENT_DA : in STRING);
```

The value for each of the fields of user-defined attributes is set when the SIMPLE_OBJECT is created.

```
procedure CREATE
  (FH : in out FILE_HANDLE;
   NAME : in STRING;
   MODE : in FILE_MODE);
```

9. Write a new value for a user-defined attribute of a given database object.

EDP:

```
procedure WRITE_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_NAME : in STRING;
   VALUE : in ATTRIBUTE_VALUE);
```

ALS: not clear that you can

AIE:

```
procedure SET_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_LABEL : in STRING;
   ATTRIBUTE_VALUE : in STRING);
```

10. Read a user-defined attribute for a given database object.

EDP:

```
procedure READ_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_NAME : in STRING;
   VALUE : out ATTRIBUTE_VALUE);
```

ALS: If you can't create/modify one, you can't read one.

AIE:

```
function GET_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_LABEL : in STRING) return STRING;
```

5.3.2 Standard and Message Output

EDP:

```
procedure PUT_STANDARD
  (TEXT_LINE : in STRING);
procedure PUT_MESSAGE
  (TEXT_LINE : in STRING);
```

ALS and AIE:

Both services are provided by PUT from package TEXT_IO. The appropriate designator for Standard_output and Message_output may either be identified explicitly as the OUT_FILE for a use of PUT, or the default designator may be assigned the appropriate value (initial default is of course for Standard_output).

5.3.3 Tool Invocation and Communication Services

1. Pass a Command Language sequence to the APSE CLI for processing, and receive information regarding the completion of the processing.

EDP:

```
procedure PROCESS_COMMANDS
  (COMMAND_SEQUENCE : in STRING;
   RETURNED_MESSAGE : out STRING;
   RESULT : out COMPLETION_STATUS);
```

ALS:

```
procedure START_LIST
  (LIST : in out PROG_DEFS.PARM_LIST_REC);
procedure MAKE_LIST
  (LIST : in out PROG_DEFS.PARM_LIST_REC;
   PARM_NAME : in KAPSE_DEFS.SHORT_ID_STRING;
   PARM_VALUE : in STRING_UTIL.VAR_STRING_REC);
procedure FINISH_LIST
  (LIST : in out PROG_DEFS.PARM_LIST_REC);
procedure CALL_WAIT
  (NAME : in KAPSE_DEFS.SHORT_ID_STR;
   PROGRAM_FILE : in KAPSE_DEFS.NODE_NAME;
   PARM_LIST : in PROG_DEFS.PARM_LIST_REC;
   STDIN_FILE : in KAPSE_DEFS.NODE_NAME;
   STDOUT_FILE : in KAPSE_DEFS.NODE_NAME;
   MSGOUT_FILE : in KAPSE_DEFS.NODE_NAME;
   PROGRAM_STATUS : in out PROG_DEFS.CALL_STATUS_REC);
```

AIE:

```
function CALL_PROGRAM
  (PROGRAM_PATH : in STRING;
   PARAMETERS : in STRING;
   CONTEXT_NAME: in := "SUB_PROGRAM_CONTEXT")
  return STRING; -- Returns output parm list
```

2. Return information regarding the completion of an invocation of Configure to the process that invoked it.

EDP:

```
procedure RETURN_STATUS
  (RESULT : in COMPLETION_STATUS;
   MESSAGE : in STRING);
```

ALS:

```
procedure RETURN_WAIT
  (RESULT_STATUS : in PROG_DEFS.RES_STATUS_INT;
   RETURN_STRING : in STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
procedure SET_ATTRIBUTE
  (OBJECT_NAME : in STRING;
   ATTRIBUTE_LABEL : in STRING;
   ATTRIBUTE_VALUE : in STRING);
```

3. Retrieve parameters passed to Configure.

EDP:

```
function TOTAL_ARG_NUMBER return INTEGER;
function KEYWORD_ARG_COUNT return INTEGER;
function POSITIONAL_ARG_COUNT return INTEGER;
function GET_KEYWORD_ARG
  (ARGNAME : in STRING)
  return STRING;
function GET_POSITIONAL_ARG
  (ORDINAL_NUMBER : in INTEGER)
  return STRING;
```

ALS:

```
procedure GET_LOCAL_STRING
  (STRING_NAME : in KAPSE_DEFS.SHORT_ID_STRING;
   STRING_VALUE : in out STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
function GET_ATTRIBUTE
  (NAME : in STRING;
   ATT_LABEL : in STRING)
  return STRING;
  Parm string is an attribute for the created program
  context (i.e GET_ATTRIBUTE(".",PARAMETERS);
function PICK_PARAM
  (PARAMETERS: in STRING;
   PARAM_NAME : in STRING;
   POSITION : in INTEGER := 0;
   DEFAULT : in STRING := "")
  return STRING;
```

APPENDIX A - REFERENCES

The following documents are references for the Interim Report on Interface Analysis.

1. Ada Integrated Environment Draft System Specification, Prepared for Rome Air Development Center, Intermetrics, 15 March 1981.
2. Ada Integrated Environment Interim Technical Report, Prepared for Rome Air Development Center, Intermetrics, 15 March 1981.
3. Ada Language System Command Language Processor B5 Specification, U.S. Army, CECOM, Ft. Monmouth, NJ, Contract No. DAAK80-80-C-0507, Draft Document CR-CP-0059-B80, August 1981.
4. Ada Language System Kernel Ada Programming Support Environment B5 Specification, U.S. Army, CECOM, Ft. Monmouth, NJ, Contract No. DAAK80-80-C-0507, Draft Document CR-CP-0059-B83, August 1981.
5. Ada Language System Specification, U.S. Army CECOM, FT. Monmouth, NJ, Contract No. DAAK80-80-C-0507, Draft Document CR-CP-0059-A000, June 1981.
6. Buxton, J.M., "Objects, Versions, Configuration, and Partitions", Unpublished Paper.
7. Configuration Management System (CMS) Program Performance Specification (PPS), Contract N00123-80-D-0364, CDRL A0001, 20 August 1982.
8. Draft Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5, Prepared for Rome Air Development Center, Intermetrics, 13 March 1981.
9. Draft Computer Program Development Specification for Ada Integrated Environment: MAPSE Command Processor Type B5, Prepared for Rome Air Development Center, Intermetrics, 13 March 1981.

10. Draft Computer Program Development Specification for Ada Integrated Environment: MAPSE Generation and Support Type B5, Prepared for Rome Air Development Center, Intermetrics, 13 March 1981.
11. Draft Computer Program Development Specification for Ada Integrated Environment: Program Integration Facilities Type B5, Prepared for Rome Air Development Center, Intermetrics, 13 March 1981.
12. Requirements for Ada Programming Support Environment, STONEMAN, February 1980.

APPENDIX B

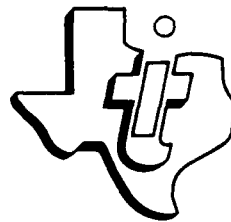
APSE
Interactive Monitor

Interim Report
on
Interface Analysis
and
Software Engineering Techniques

Prepared for

NAVAL OCEAN SYSTEMS CENTER
United States Navy

Contract No. N66001-82-C-0440
CDRL Seq. No. A011



Equipment Group - ACSL
P.O. Box 405, M. S. 3407
Lewisville, Texas 75067
16 May 1983

TEXAS INSTRUMENTS
INCORPORATED

TABLE of CONTENTS

Paragraph	Title	Page
-----------	-------	------

SECTION 1 INTRODUCTION

1.1	Purpose	1-1
1.2	Background	1-1

SECTION 2 ACKNOWLEDGEMENTS

SECTION 3 AIM INTERFACE REQUIREMENTS

3.1	General	3-1
3.2	Terminal Communication Services Interfaces	3-1
3.2.1	Terminal Capabilities	3-1
3.3	APSE Program Control and Communication Interfaces	3-5
3.3.1	Program Control Interfaces	3-5
3.3.2	Interprocess Communication	3-5
3.4	KAPSE Database Interfaces	3-6
3.4.1	Summary of Database Interfaces	3-6
3.5	Miscellaneous Interfaces	3-7
3.5.1	Date information	3-7
3.5.2	Call-Tree Information	3-7

SECTION 4 ADA LANGUAGE SYSTEM INTERFACES

4.1	General	4-1
4.2	Terminal Communication Services Interfaces	4-1
4.2.1	Sufficient Interfaces	4-1
4.2.2	Insufficient Interfaces	4-1
4.3	APSE Program Control and Communication Interfaces	4-3
4.3.1	Sufficient Interfaces	4-3
4.3.2	Insufficient Interfaces	4-5
4.4	KAPSE Database Interfaces	4-5
4.4.1	Sufficient Interfaces	4-5
4.5	Miscellaneous Interfaces	4-7

AD-A141 576

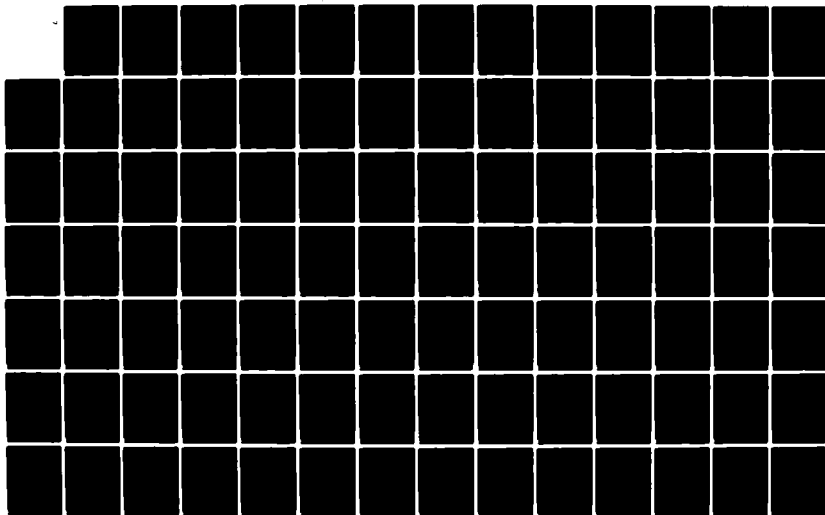
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM PUBLIC REPORT VOLUME 3(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 25 OCT 83
NOSC/TD-552-VOL-3

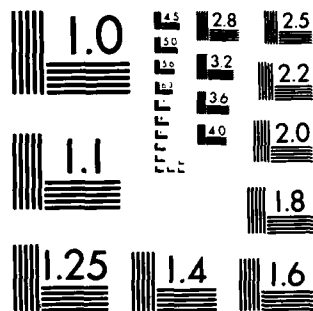
46

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4.5.1	Sufficient Interfaces	4-7
-------	---------------------------------	-----

SECTION 5 ADA INTEGRATED ENVIRONMENT INTERFACES

5.1	General	5-1
5.2	Terminal Communication Services Interfaces	5-1
5.2.1	Sufficient Interfaces	5-1
5.2.2	Insufficient Interfaces	5-3
5.3	APSE Program Control and Communication Interfaces	5-3
5.3.1	Sufficient Interfaces	5-3
5.3.2	Insufficient Interfaces	5-4
5.4	KAPSE Database Interfaces	5-7
5.4.1	Sufficient Interfaces	5-7
5.5	Miscellaneous Interfaces	5-8
5.5.1	Sufficient Interfaces	5-8
5.5.2	Insufficient Interfaces	5-8

SECTION 6 STANDARD INTERFACE SET

6.1	General	6-1
6.2	Background	6-1
6.3	Terminal Communication Services Interfaces	6-1
6.3.1	Sufficient Interfaces	6-1
6.3.2	Insufficient Interfaces	6-2
6.4	APSE Program Control and Communication Interfaces	6-3
6.4.1	Sufficient Interfaces	6-3
6.4.2	Insufficient Interfaces	6-5
6.5	KAPSE Database Interfaces	6-6
6.5.1	Sufficient Interfaces	6-6
6.6	Miscellaneous Interfaces	6-7

SECTION 7 AIM PORTABILITY ISSUES

7.1	General Portability Issues	7-1
7.2	AIM Environment Dependencies	7-1
7.3	AIM Environment Dependent Areas	7-1
7.3.1	Terminal Communication Services	7-1
7.3.2	APSE Program Interfaces	7-3
7.3.3	Database interfaces	7-8
7.3.4	Miscellaneous interfaces	7-11
7.4	KAPSE Document Quality	7-12
7.4.1	Document Deficiencies	7-13

SECTION 8 USER INTERFACES

8.1	Constraints on User Programs	8-1
8.1.1	APSE Program I/O	8-1
8.1.2	MASTER IN, MASTER OUT, and MESSAGE OUT	8-2
8.2	Command Language Processor Constraints	8-2

SECTION 9 KAPSE ISSUES

9.1	General	9-1
9.2	Bypassing KAPSE Services for Program Control	9-1
9.2.1	ALS "Break-In" facility	9-1
9.2.2	AIE "Scroll Mode Control"	9-2
9.3	Broadcast Messages	9-2
9.4	Bypassing KAPSE Services for Terminal Control	9-2
9.5	Programs targeted for specific terminals	9-3

APPENDIX A AIM INTERFACES SUMMARY

A.1	Interface Comparison	A-1
A.2	Interface Summary	A-3

APPENDIX B ARPANET COMMUNICATIONS

APPENDIX C GLOSSARY

APPENDIX D REFERENCES

D.1	Government Standards	D-1
D.2	Government Specifications	D-1
D.3	Other Government Documents	D-3
D.4	Special Sources	D-3
D.5	Other Publications	D-3

SECTION 1

INTRODUCTION

1.1 Purpose

An Ada* program requires clear and well-defined interfaces to interact with an Ada Programming Support Environment (APSE). To meet the goals of interoperability and transportability, an Ada program intended to execute under more than one APSE should interface equally well with each system. Accomplishing this communication entails study of existing APSE features mapped against the requirements of the Ada program. Through analysis, APSE interface strengths and deficiencies are revealed.

This interim report is an analysis of the APSE Interactive Monitor (AIM) presently under development by Texas Instruments under NOSC contract N66001-82-C-0440. AIM requirements are mapped against the designed features of two APSEs: the U.S. Army Ada Language System (ALS) and the U.S. Air Force Ada Integrated Environment (AIE). Also considered in the study is the Standard Interface Set (SIS) currently under development by the KAPSE Interface Team (KIT). Since neither the ALS, the AIE, nor the SIS is complete at the time of this report, the analysis is preliminary. The information herein is the most current available; future changes in the aforementioned systems may affect AIM interfaces.

1.2 Background

The APSE Interactive Monitor (AIM) is a tool designed to act as an interface between the user of the APSE and the programs the user executes in the APSE. It enables a user to execute multiple APSE programs from a single terminal while keeping their interactive inputs and outputs separate both logically and physically. The standard terminal and program control facilities available through the APSE will be temporarily supplemented or replaced by the AIM during its execution. For a complete description of AIM functionality, consult [TIR3].

* [Ada is a registered trademark of the U.S. Department of Defense, Ada Joint Program Office.]

The primary objective of the AIM project is to assist the KAPSE Interface Team (KIT) in studying ALS and AIE interface issues while secondarily producing a useful tool for APSEs. The reader should be familiar with the AIM Program Performance Specification [TI83], the ALS System Specification [SOF82], and the AIE System Specification [INT82].

SECTION 2

ACKNOWLEDGEMENTS

The APSE Interactive Monitor (AIM) is under development by Texas Instruments, Inc., under U.S. Navy contract number N66001-82-C-0440. John Foreman is the project manager for this effort. Preliminary design for the AIM was accomplished by Tim Harrison and Stewart French. The research for this report was performed by Tim Harrison, Stewart French, and Melody Moore. This report was written by Melody Moore in collaboration with the other members of the AIM design team.

SECTION 3

AIM INTERFACE REQUIREMENTS

3.1 General

This section outlines the interfaces the AIM requires of the KAPSEs and the rationale behind them. Described below are general issues and ideal solutions to AIM implementation problems. The next three sections of the report parallel this section, relating AIM requirements to ALS, AIE, and SIS provisions.

The AIM requires well-defined interfaces in the following areas:

1. Terminal Communication
2. APSE Program Control and Communication
3. Database Services
4. Miscellaneous KAPSE Services
 - a. Date
 - b. Call-tree information

3.2 Terminal Communication Services Interfaces

The AIM interacts with scroll mode ("TTY"), page mode (screen-oriented), and form mode (IBM 3270-type) physical terminals. The capabilities of a terminal intended for use with the AIM must be a functionally compatible subset of the standard capabilities described in [ANSI79].

3.2.1 Terminal Capabilities. Scroll mode terminals are line-oriented and possess a very limited set of capabilities. Generally, characters are transmitted and received one at a time, or as a line block delimited by a line feed and/or carriage return.

Page mode terminals are screen-oriented and possess extended two-dimensional functional capabilities. Characters are transmitted and received one at a time.

Form mode terminals are also screen-oriented with two-dimensional capabilities, but data is transferred in blocks delimited by a keystroke (such as "enter") instead of one character at a time. Simple editing functions can be performed by the terminal itself without host computer interaction. Some form mode terminals possess the capability to write-protect selected screen areas.

Terminals that provide the facilities of all three types (scroll, page, and form mode) do exist, but for AIM purposes, the functionalities are considered separately.

The AIM requires the following support from the KAPSE terminal communication services:

1. The ability to read single characters from the keyboards of terminals which support this functionality with no automatic I/O buffering.

Rationale: The AIM belongs to a group of highly interactive programs which ideally should have access to characters immediately as they are generated by the keyboard. In order to afford maximum control of terminal I/O to these programs, there should be no interpretation of key sequences entered at the host OS or KAPSE level; interactive programs provide their own interpretation. Buffering usually implies an interpretation of at least one character (the "end of buffer" mark, usually carriage return). This rationale does not preclude buffering which supports the desirable type-ahead capability. Characters may be piped into a channel to wait for reading and still be accessible singly.

In addition to the AIM, programs such as screen editors and powerful interactive command language interpreters require character-by-character input for implementation. This need is further supported in [COX83]:

"Since it would be unreasonable to make it impossible to implement screen oriented text editors or advanced command line interpreters in the APSE, immediate acquisition of input characters must be provided. The Ada language definition avoids this issue and leaves the Ada programmer at the mercy of side effects arising from system buffering of I/O. This I/O facility should therefore be provided to Ada programmers through the KAPSE interface." [COX83]

I/O buffering also affects the AIM updating mechanism. The cursor is always positioned at the current read or write location on the screen. The AIM updates several viewports asynchronously while allowing the user to enter keystrokes destined for a particular APSE program's input. Consequently, the AIM requires the freedom to move the cursor immediately to any location where an I/O transaction is destined to occur. If characters are buffered by the terminal (i.e., transmitted only on depression of carriage return), the AIM waits for the input of an entire character string before releasing the cursor for I/O elsewhere on the screen. This buffering would limit the AIM asynchronous screen updating mechanism, because the AIM would not receive each character as it was generated.

2. The ability to enable and disable character echo on the display as characters are input from the terminal keyboard.

Rationale: The argument for screen echo control is similar to the previous rationale for single-character read and write. Screen echo may be controlled from a variety of sources:

- a. Directly in the terminal (acceptable for form mode terminals)
- b. From a communication line (i.e., modem)
- c. From the KAPSE
- d. From the host operating system
- e. From an APSE program (required for terminals capable of transmitting a character at a time)

To permit asynchronous update and to control the screen display, the AIM requires the ability to disable echo in order to place characters in the correct screen location. For example, if the cursor is involved in a write operation at the top of the screen, and a character intended to follow a command prompt at the bottom of the screen is typed from the keyboard, the newly-input character could be echoed in the middle of the write transmission unless the AIM can disable character echo. The AIM itself should receive the character and echo it in the appropriate place. The KAPSE terminal communication services should permit echo disabling when possible in order to facilitate this diverse control.

It should also be noted that some interactive editing tools (such as EMACS) cannot be implemented unless echo disabling

and non-buffered I/O are provided. These tools must also be able to intercept data without echo in order to control the display of data on the screen.

3. The ability to obtain exclusive access to the user's terminal.

Rationale: The AIM needs to be able to manage all data destined for, or received from, the terminal. If the AIM cannot protect the screen from other programs' I/O, any APSE program could write to the screen, perhaps causing undesirable data transmissions to collide with or overwrite AIM transmissions. Since the AIM maintains an internal representation of the screen, a re-write facility could be provided; however, constantly rewriting the screen would be inconvenient for the user.

4. The ability to write variable length character strings to the terminal display device exactly as they are represented in the generating program.

Rationale: The AIM must be able to write one or more characters to the screen with no extraneous characters (such as line feed) automatically appended, and no character translation operations occurring. Terminals are controlled by specific protocols transmitted by the AIM and interpreted by the display device. Terminal communication protocols consist of character sequences in which the relationships between specific characters are given meaning. Adding or removing any character in a sequence may alter its meaning.

5. The ability to obtain specific information concerning the terminal's capabilities and features.

Rationale: Some standard method of naming terminal types must exist to enable terminal capabilities to be mapped into a database file. The AIM must be able to retrieve this information and identify the correct physical terminal in order to initialize the correct logical terminal. If the KAPSE does not provide this feature, the AIM may ask the user to identify the type of terminal to initialize.

6. Screen-oriented facilities.

Rationale: The AIM requires the control to position or move the cursor and perform simple editing functions on its two dimensional display. With these minimum capabilities, the AIM can simulate other more complicated editing functions such as insert or delete line.

3.3 APSE Program Control and Communication Interfaces

3.3.1 Program Control Interfaces. The AIM interfaces with the KAPSE program control facilities to govern the execution of specified APSE programs and their corresponding interprocess communication. The AIM requires the following capabilities:

1. Initiate a new program
2. Abort program
3. Suspend program
4. Resume program
5. Determine program status

Rationale: The AIM controls APSE programs and subordinate programs spawned from these programs through this interface. The AIM user may invoke APSE programs from the AIM and control their execution. The above program control functions are defined as basic APSE requirements in "STONEMAN" [DOD80] and therefore should be common to all APSEs.

3.3.2 Interprocess Communication. Interprocess Communication is an important AIM interface. The AIM "captures" terminal-directed output from APSE programs and program-directed input from the terminal.

Rationale: The AIM requires that all data be received in the same order as it was transmitted, in essence, a data "pipe". This pipe scheme should be totally transparent to the APSE program running under the AIM; the AIM should not in any way affect the implementation of APSE program I/O.

Since an APSE program is defined as one which only uses KAPSE services, The AIM requires APSE programs to use only Standard In and Standard Out for terminal directed I/O. An APSE program which bypasses the KAPSE to use underlying host services is considered erroneous, and may produce undesirable results when executed under the AIM. Only one terminal may be associated with an APSE program, since there is no method of identifying multiple terminals. (See section 7, "User Interfaces".)

3.4 KAPSE Database Interfaces

The AIM manipulates database files during initialization and execution. The packages TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO defined in [DOD82] provide the AIM with clear interfaces for file manipulation. Both APSEs augment the I/O capabilities defined in the Ada language with extended features.

3.4.1 Summary of Database Interfaces. The following is a list of AIM functional requirements which the KAPSE database services should fulfill:

1. Open a file for reading/writing
2. Read from a file
3. Write to a file
4. Close a file
5. Create/delete a file

Rationale: During its initialization, the AIM reads information from several specific KAPSE database files. When invoked, the AIM attempts to read commands from a predefined indirect command script. The KAPSE database services must provide for opening and reading this indirect command script.

The AIM also requests KAPSE database service support for accessing the predefined Terminal Capabilities File. This file describes common terminal functions in terms of device-specific control sequences, allowing the AIM to manipulate many different physical terminals with a common interface. It is possible to update terminal capabilities; therefore, this file must also be modifiable.

The AIM also requires KAPSE database services during execution. In order to log all terminal-destined I/O to permanent files, the AIM provides an optional "pad" feature. A pad consists of two KAPSE database files which mirror all input and output transferred between an APSE program executing in the environment and its associated AIM window. Each APSE program mapped to an AIM window may have its own pad. The AIM requires a method of creating, deleting, and controlling pad files from the KAPSE database services.

3.5 Miscellaneous Interfaces

3.5.1 Date information.

Rationale: The AIM requires KAPSE services to obtain the current date to display in the viewport header.

3.5.2 Call-Tree Information.

Rationale: The AIM queries the KAPSE program call-tree structure to determine subordinate program information. The AIM examines this structure to manage programs executing under its control. For example, the user may not exit the AIM unless all subordinate programs have terminated. The AIM determines which programs and subprograms are currently executing by examining the call tree.

SECTION 4

ADA LANGUAGE SYSTEM INTERFACES

4.1 General

The Ada Language System (ALS) is under development by SofTech, Inc. under U. S. Army contract number DAAK80-80-C-0507. The implementation of the ALS is not yet complete.

4.2 Terminal Communication Services Interfaces

4.2.1 Sufficient Interfaces. The following ALS terminal communications interface is satisfactory and well-defined for the AIM requirements:

1. Write variable length strings to user terminal. This capability is provided in TEXT_IO by the following procedure: ([DOD82] p 14-9)

```
procedure PUT (FILE : in FILE_TYPE;  
              ITEM : in STRING) ;
```

4.2.2 Insufficient Interfaces. There are many interfaces in the terminal communications area that are missing or not well defined in the ALS. The following are the AIM interface requirements which are nonexistent or undefined in the ALS:

1. Read single character - not provided.

According to ARPANET response from SofTech, the ALS KAPSE does not perform any I/O buffering. However, the underlying host (VMS) buffers keyboard input except for special control character sequences which cause interruption (such as <CNTRL>-Y). The ALS provides no mechanism for bypassing this VMS buffering. (See Appendix B question 3.)

The VMS terminal driver waits for a carriage return to transmit characters. The AIM can circumvent buffering of character I/O by using the VAX QIO macros [DEC82]. This is a significant interface issue because it requires bypassing the ALS KAPSE. Consequently, the portability of the AIM is reduced.

An issue closely related to reading single characters without buffering is writing a single character. The ALS implementation of the Ada package TEXT_IO also buffers output. Characters are transmitted only upon execution of NEW LINE, PUT_LINE, or when the line length is exceeded. It is possible to set the line length to 1 to transmit one character at a time; however, the "end of line" mark would be inserted after each character.

2. Enable/disable echo - not provided.

The ALS defines only two packages to supplement Ada file I/O: packages BASIC_IO and AUX_IO; there is no TERMINAL_IO package. Neither of the defined packages support echo enable/disable.

3. Exclusive access to user terminal- not provided.

The ALS documents were not clear on the subject of user terminal access. Verbal response from SofTech [RT83] indicated that exclusive access to the user terminal is not provided by the ALS.

4. Write exactly as internally represented - not provided.

The ALS KAPSE does no character translations of its own. Package BASIC_IO provides byte string read and write from the KAPSE to the VMS terminal driver. Again, however, the ALS does not allow the user to control the VMS device driver through the KAPSE. (See Appendix B, question 1.)

The VMS terminal driver has the potential to perform character string translations. For example, if a terminal uses an 8-bit ASCII character code and the TTSM_EIGHTBIT mode is not set, the device driver assumes a seven-bit code, masking out the eighth bit (dropping a bit from the received byte). Clearly, this behavior could alter an AIM transmission. VMS also allows syntax validation of escape sequences if TTSM_ESCAPE mode is set, which forces certain interpretations of AIM control sequences. The ALS KAPSE provides no services for setting these VMS terminal characteristic modes; the user must perform an ESCAPE to the underlying VMS operating system.

To write strings exactly as represented, the ALS KAPSE may be bypassed to access the VMS device driver and set the terminal characteristic TTSM_PASSALL. This mode ensures that all input and output is binary and that no interpretation whatsoever occurs in the device driver. Again, AIM transportability is significantly reduced by the VMS dependent services required. ([DEC82] p 9-19)

5. Terminal identification - not provided.

The ALS documents do not describe a method for obtaining terminal identification, and verbal communication confirmed that this AIM requirement is not supported. [RT83]

6. Screen-oriented facilities - not provided

As described above, the ALS defines only two auxiliary I/O packages to supplement Ada I/O. Neither of these packages mentions any screen manipulation procedures, and ARPANET communications confirmed that explicit x-y cursor positioning is not supported by the ALS. (Appendix B, question 2.)

4.3 APSE Program Control and Communication Interfaces

4.3.1 Sufficient Interfaces. The AIM program control requirements are supported by the ALS as follows:

1. Initiate program - Procedure CALL_WAIT is provided in package PROG_CALL to invoke a program and wait until the program completes its execution. Procedure CALL_NO_WAIT invokes a new APSE program allowing the caller to continue execution without waiting for the invoked program to complete: ([SOF82] p 90-138, [SOF82A])

```

procedure CALL_WAIT (PROGRAM_NAME : in
                     KAPSE_DEFS.SHORT_ID_STRING;
                     PROGRAM_FILE : in KAPSE_DEFS.NODE_NAME;
                     PARAMETER_LIST : in
                     PROG_DEFS.PARM_LIST_REC;
                     STDIN_FILE : in KAPSE_DEFS.NODE_NAME;
                     STDOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                     MSGOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                     PROGRAM_STATUS : in out
                     PROG_DEFS.CALL_STATUS_REC);

```

```

procedure CALL_NO_WAIT (PROGRAM_NAME : in
                        KAPSE_DEFS.SHORT_ID_STRING;
                        PROGRAM_FILE : in KAPSE_DEFS.NODE_NAME;
                        PARAMETER_LIST : in
                        PROG_DEFS.PARM_LIST_REC;
                        STDIN_FILE : in KAPSE_DEFS.NODE_NAME;
                        STDOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                        MSGOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                        PROGRAM_STATUS : in out
                        PROG_DEFS.CALL_STATUS_REC);

```

2. Control of APSE programs from within other APSE programs.
 Package PROG_CONTROL defines program control procedures to suspend execution of an APSE program and all of its subprogram descendants, to restart a specified APSE program, to abort a program and all of its subprogram descendants, and to request program status:

- a. Suspend program ([SOF82] p 90-156)

```

procedure REQ_SUSPENSION (PROG_NAME : in
                          STRING_UTIL.VAR_STRING_REC;
                          REQUEST_STATUS : out
                          PROG_DEFS.PROCAL_STATUS_ENU) ;

```

- b. Resume program ([SOF82] p 90-162)

```

procedure REQ_RESUMPTION (PROG_NAME : in
                          STRING_UTIL.VAR_STRING_REC;
                          REQUEST_STATUS : out
                          PROG_DEFS.PROCAL_STATUS_ENU) ;

```

- c. Abort program ([SOF82] p 90-161)

```

procedure REQ_ABORTION (PROG_NAME : in
                       STRING_UTIL.VAR_STRING_REC;
                       REQUEST_STATUS : out
                       PROG_DEFS.PROCAL_STATUS_ENU);

```

- d. Determine program status ([SOF82 p 90-157)

```

procedure REQ_STATUS (PROG_NAME : in
                     STRING_UTIL.VAR_STRING_REC;
                     PROGRAM_STATUS : out
                     PROGRAM_INFO_REC) ;

```

4.3.2 Insufficient Interfaces. The APSE program interfaces which are not sufficient in the ALS are:

1. Intercept APSE program's terminal I/O - Not provided.

There are no provisions in the ALS for intercepting an APSE program's terminal I/O. To support this requirement, the AIM may have to bypass the ALS KAPSE to access underlying VMS mailboxes. The VAX Create Mailbox and Assign Channel (SCREMBX) system allows processes to create channels for terminal I/O, which may allow the AIM to intercept an APSE program's I/O destined for the terminal. ([DEC82] p 8-2) Naturally, transportability is impaired whenever the KAPSE is bypassed.

2. Exclusive access to the APSE program's terminal I/O - not provided

The ALS defers to VMS which allows concurrent read and write access to multiple internal files which are associated with the terminal. The KAPSE provides no mechanism for obtaining exclusive access to the APSE program's terminal I/O (Appendix B, question 4). VMS, however, provides a device allocation service (\$ALLO) which reserves the device for the exclusive use of the requesting process and its subprocesses. Again, this requires bypassing the ALS KAPSE.

3. Interprocess communication - not provided.

The ALS makes no provisions for this capability. There is also no way to open an I/O file in SHARED_STREAM mode, as the AIE describes. As described above, the ALS KAPSE can be bypassed to access VMS mailboxes to accomplish this communication, which hinders transportability.

4.4 KAPSE Database Interfaces

4.4.1 Sufficient Interfaces. The ALS KAPSE Database Services are complete and sufficient for AIM implementation. The ALS replaces and augments the file manipulation capabilities defined in TEXT_IO [DOD82] with the package BASIC_IO, which contains procedures to control the standard ALS I/O streams. Although the TEXT_IO provisions are sufficient for AIM needs, BASIC_IO enhances I/O for the ALS interface. The AIM database interface requirements are fulfilled as follows:

1. Open/Close a database file

The ALS provides open and close procedures in package BASIC_IO: ([SOF82] p 90-50,53)

```
procedure OPEN_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;  
                     NAME : in STRING_UTIL.VAR_STRING_REC;  
                     MODE : in IO_DEFS.IO_MODE_ENU;  
                     FILE_CLASS : out IO_DEFS.FILE_CLASS_ENU;  
                     RECORD_FORMAT : out  
                       IO_DEFS.RECORD_FORMAT_ENU;  
                     RECORD_LENGTH : out  
                       IO_DEFS.DATA_INDEX_INT;  
                     RESULT : out IO_DEFS.IO_RESULT_ENU;  
                     RESULT_STRING : in out  
                       STRING_UTIL.VAR_STRING_REC) ;
```

```
procedure CLOSE_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;  
                     RESULT : out IO_DEFS.IO_RESULT_ENU;  
                     RESULT_STRING : in out  
                       STRING_UTIL.VAR_STRING_REC) ;
```

2. Read/write to a database file

Package BASIC_IO in the ALS supports procedure READ_FILE which reads data from an open input file. Similarly, procedure WRITE_FILE writes data to an open output file: ([SOF82] p 90-55,-57)

```
procedure READ_FILE (STREAM : in IO_DEFS.STREAM_ID_PRIV;  
                   BUFFER : in KAPSE_DEFS.ADDRESS_INT ;  
                   LENGTH : in IO_DEFS.DATA_INDEX_INT ;  
                   COUNT : out IO_DEFS.DATA_LENGTH_INT ;  
                   RESULT : out IO_DEFS.IO_RESULT_ENU ;  
                   RESULT_STRING : in out  
                     STRING_UTIL.VAR_STRING_REC) ;
```

```
procedure WRITE_FILE (STREAM : in IO_DEFS.STREAM_ID_PRIV;  
                    BUFFER : in KAPSE_DEFS.ADDRESS_INT ;  
                    LENGTH : in IO_DEFS.DATA_INDEX_INT ;  
                    COUNT : out IO_DEFS.DATA_LENGTH_INT ;  
                    RESULT : out IO_DEFS.IO_RESULT_ENU ;  
                    RESULT_STRING : in out  
                      STRING_UTIL.VAR_STRING_REC) ;
```

3. Create database files

File creation is also handled in package BASIC_I/O by procedure MAKE_FILE ([SOF82] p 90-47):

```
procedure MAKE_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;
                     NAME : in STRING_UTIL.VAR_STRING_REC;
                     MODE : in IO_DEFS.IO_MODE_ENU ;
                     FILE_CLASS : in IO_DEFS.FILE_CLASS_ENU;
                     RECORD_LENGTH : in IO_DEFS.DATA_INDEX_INT;
                     RESULT : out IO_DEFS.IO_RESULT_ENU;
                     RESULT_STRING : in out
                       STRING_UTIL.VAR_STRING_REC) ;
```

This procedure associates an I/O stream with the new file.

4. Destroy database files

Package BASIC_IO also supports file deletion by procedure DELETE_FILE: ([SOF82] p 90-49)

```
procedure DELETE_FILE (STREAM : in IO_DEFS.STREAM_ID_PRIV;
                      RESULT : out IO_DEFS.IO_RESULT_ENU;
                      RESULT_STRING : in out
                        STRING_UTIL.VAR_STRING_REC) ;
```

The database file associated with the STREAM parameter is deleted, after all streams with which the file is associated are closed.

4.5 Miscellaneous Interfaces

4.5.1 Sufficient Interfaces.

1. Date information.

In addition to the predefined Ada package CALENDAR [DOD82], the ALS defines procedure GET_TIME_DATE in package MISC_SERV ([SOF82] p 90-184) to retrieve the current time and date in standard ALS format:

```
procedure GET_TIME_DATE (TIME_INFO : out
                        MISC_DEFS.TIME_INFO_REC;
                        DATE_INFO : out
                        MISC_DEFS.DATE_INFO_REC) ;
```

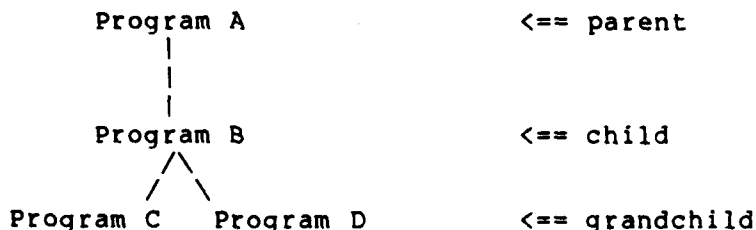
2. Call-tree information.

Package `PROG_CONTROL` describes procedure `REQ_STATUS` which provides the capability to query the call tree: ([SOF82] p 90-157)

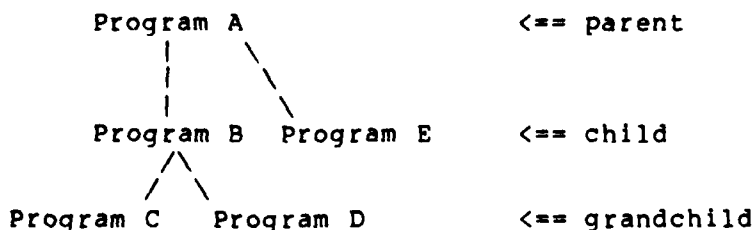
```
procedure REQ_STATUS (PROG_NAME : in
                     STRING_UTIL.VAR_STRING_REC;
                     PROGRAM_STATUS : out PROGRAM_INFO_REC) ;
```

The `PROGRAM_STATUS` parameter is a record which contains descendant program names. The user may repeatedly call `REQ_STATUS` to retrieve names one by one, in effect traversing the call-tree for information. ([SOF81B] p 3-174)

This approach to information retrieval has some interesting implications. The call-tree is a dynamic structure. It is not clear if the `REQ_STATUS` procedure takes a "snapshot" of the call-tree at a given instant, or (more likely) the call-tree continues changing as the status of various programs also change. A user that repeatedly calls `REQ_STATUS` to traverse the call-tree has no way of assuring that the information retrieved in one instant will be valid the next instant. For example, if the user wishes to count the total number of subordinate programs of a running program "A", a call to `REQ_STATUS` would indicate one child program, "B", directly under "A":



The user would then call `REQ_STATUS` again to determine if program "B" has any subordinate programs. While the user is querying program "B", program "A" could start another program, "E":



The user will never know about program "E" unless he or she performs another REQ_STATUS on program "A". Therefore, the program count will probably be invalid.

Fortunately, the AIM requires call-tree information to determine simply whether or not a program has subordinate programs, so repeated calls to REQ_STATUS and the possible problems described above are not anticipated.

SECTION 5

ADA INTEGRATED ENVIRONMENT INTERFACES

5.1 General

Intermetrics, Inc. is performing the design and implementation of the Ada Integrated Environment (AIE) under U. S. Air Force contract number F30602-80-C-0291. Neither the design nor the implementation of the AIE are complete at this time.

5.2 Terminal Communication Services Interfaces

5.2.1 Sufficient Interfaces. The following AIM terminal communication interfaces are sufficient for AIM requirements and are well-defined in the AIE:

1. Enable/disable echo.

The AIE provides package INTERACTIVE_IO as an extension to TEXT_IO. The following procedures are defined in this package for echo control:

```
procedure SET_ECHO (INPUT : in FILE_TYPE ;  
                   OUTPUT: in FILE_TYPE) ;
```

```
procedure NO_ECHO (INPUT : in FILE_TYPE) ;
```

```
procedure NO_ECHO (OUTPUT: in FILE_TYPE) ;
```

SET_ECHO enables echo at the current line and column of input. NO_ECHO breaks echo associations for either input or output. ([INT82] p 73)

2. Exclusive access to the user terminal.

Although this requirement is not discussed in the documents, verbal communication with Intermetrics indicated that there will exist a method for obtaining exclusive access to the user terminal. [TT83]

3. Write to terminal exactly as internally represented.

Verbal communication [TT83] confirmed that character sequences may be sent to the terminal with no intervening interpretation or character additions or deletions.

4. Write variable length strings.

This functionality is defined in the package TEXT_IO by the following procedure: ([DOD82] p 14-19)

```
procedure PUT (FILE : in FILE_TYPE; ITEM : in STRING) ;
```

5. Terminal identification.

Package TERMINAL_IO is defined in the AIE for terminal interaction. The following primitives could potentially retrieve a terminal identification string: ([INT82] p 58)

```
procedure GET_TERMINAL_INFO (TERM : in INTEGER ;  
                             INFO : out TERMINAL_INFO_BLOCK);
```

If TERMINAL_INFO_BLOCK (yet undefined) contains a component such as TERMINAL_ID_STRING, these procedures would suffice for this interface requirement. Verbal communication [TT83] confirmed this assumption.

6. Screen-oriented facilities

Since the AIE treats the terminal display as a random text file, the package TEXT_ACCESS defines several procedures applicable to two-dimensional display manipulation. The AIM can simulate all the screen "editing" functions it requires with the primitives in this package. For example, a "clear to end of line" command can be implemented by positioning the cursor to the appropriate line and column, and writing blanks across the line. Included in package TEXT_ACCESS are: ([INT82] p 70)

- a. Procedure SET_OFFSET - selects the next read/write character.

```
procedure SET_OFFSET (FILE : in FILE_TYPE;  
                     TO : in COUNT);
```

- b. Procedure SET_LINE - positions the cursor vertically at the beginning of a selected line.

```
procedure SET_LINE (FILE : in FILE_TYPE;  
                    TO : in COUNT);
```

- c. Procedure SET_COL - positions the cursor horizontally at a selected column.

```
procedure SET_COL ( FILE : in FILE_TYPE;  
                   TO : in COUNT);
```

5.2.2 Insufficient Interfaces. The following are problem areas where AIE terminal communication interfaces required by the AIM were either nonexistent or not well defined in the AIE:

1. Read single character - not provided

Arpanet response from Intermetrics indicated that I/O operations to interactive devices are buffered to permit local line-editing before the characters are received as part of the text input file. (Appendix B, question 3) Buffers will be delimited by ENTER/Carriage Return characters. Projections indicate that the SET INPUT_INFO procedure of package INTERACTIVE_IO will provide control of this buffering. ([INT82] p 73)

5.3 APSE Program Control and Communication Interfaces

5.3.1 Sufficient Interfaces. The following APSE Program Interfaces are sufficiently well-defined in the AIE to support AIM requirements.

1. Control of APSE programs from within other APSE programs. The AIE defines package PROGRAM_INVOCATION to support program control requirements: ([INT82] p 105-6)
 - a. Initiate. Function CALL_PROGRAM invokes an APSE program as a subprogram of the caller. The calling program is suspended until completion of the subprogram. Procedure INITIATE_PROGRAM invokes an APSE program in a manner similar to CALL_PROGRAM, but the calling program is not suspended:

```

function CALL_PROGRAM (PROGRAM_PATH : in STRING;
                       PARAMETERS: in PARAMS_STRING;
                       CONTEXT_NAME : in STRING :=
                           ".SUB_CONTEXT";
                       STD_IN : in TEXT_IO.FILE_TYPE :=
                           CURRENT_INPUT;
                       STD_OUT : in TEXT_IO.FILE_TYPE :=
                           CURRENT_OUTPUT)
return RESULTS_STRING ;

```

- b. Suspend. Procedure SUSPEND_PROGRAM allows execution to be temporarily stopped.

```

procedure SUSPEND_PROGRAM (CONTEXT_NAME : in STRING);

```

- c. Resume. Procedure RESUME_PROGRAM restarts a suspended program.

```

procedure RESUME_PROGRAM (CONTEXT_NAME : in STRING) ;

```

- d. Abort. Procedure EXIT_PROGRAM stops program execution. A boolean parameter indicates whether to wait for subprocesses to complete, or to abort all subprocesses.

```

procedure EXIT_PROGRAM (RESULTS : in RESULTS_STRING;
                       ABORT_SUB_CONTEXTS : in boolean :=
                           false) ;

```

5.3.2 Insufficient Interfaces. The following program control interfaces are either nonexistent or not well defined in the AIE:

1. Intercept APSE program's terminal I/O - unable to determine.

The AIM desires channels for pipe I/O that are transparent to an APSE program. According to ARPANET response from Intermetrics, pipe I/O can be accomplished through package TEXT_IO, opening the file in SHARED_STREAM mode. The OPEN procedure allows a FORM string parameter which can be specified as SHARED_STREAM through a label=>value list: ([INT82] p 72) ([DOD82], 14.3.10) (Appendix B, question 4)

```

OPEN (FILENAME, IN_FILE, "RESERVE_MODE=>SHARED_STREAM") ;

```

SHARED_STREAM mode allows synchronization of database object access so that WRITE ORIGINAL access is "reserved" (granted) only at the time of the READ or WRITE. ([INT82] p 94)

This approach to pipe I/O may not be acceptable for the AIM. The AIM requires that APSE programs use only the STD_IN and STD_OUT files for terminal I/O. These files are implicitly opened upon program invocation, so normally the APSE program never calls the OPEN procedure. The AIE mechanism requires that the APSE user program explicitly open STD_IN and STD_OUT in SHARED_STREAM mode. This implies that the AIM pipe mechanism would not be transparent to user programs.

A more attractive alternative is for the AIM itself to open the file in SHARED_STREAM mode, and then pass the file to the APSE program as part of the INITIATE call. The AIM pipe mechanism would then be transparent to the APSE program. Verbal communications indicate that this is a viable solution. [TT83A]

2. Exclusive access to the APSE program's terminal I/O - unable to determine.

The AIE documents presently contain minimal information about terminal I/O. The TEXT_IO package [DOD82] defines facilities for many I/O needs, but does not define a method for connecting the STD_IN and STD_OUT files to the terminal, as required by the AIM. Therefore, it is difficult to determine whether the AIM may be granted exclusive access to an APSE program's terminal I/O.

3. Determine program status - indeterminate.

There is no explicit provision in [INT82] for querying the status of a program. Package PROGRAM_INVOCATION contains procedure SUSPEND PROGRAM, which stops the execution of the named program, "allowing the state of the execution to be examined" ([INT82] p 106). It is not clear if the state of execution returned is the current status, or the state of the program before suspension. Since it would be useless to query the status of a process that is known to be suspended, one would assume that the execution state returned consists of program information (such as register contents). Under this assumption, program status information is not available to the user.

4. Interprogram Communication.

AIM interprogram communication consists of intercepting input and output from the APSE program executing in the

environment. The AIM may be able to accomplish this with the provision for opening a file in SHARED_STREAM mode (see item 1 above).

It is not clear that the AIM will need further interprogram communications facilities, but for the sake of completeness, the existing AIE IPC interfaces are analyzed below.

The package INTER PROGRAM COMMUNICATION defines several functions and procedures for manipulating and controlling program I/O channels. The communicating programs must agree on the format and interpretation of PARAMS_STRING and RESULTS_STRING for interprogram communication. ([INT82] p. 110)

a. Accept next waiting entry call:

```
function IPC_ACCEPT (CHANNEL_NAME : in STRING;  
                    TIME_LIMIT : in DURATION  
                    := DURATION'LAST)  
return PARAMS_STRING ;
```

b. Resume IPC_ENTRY_CALL after an IPC_ACCEPT:

```
procedure IPC_END_RENDEZVOUS (CHANNEL_NAME : in STRING;  
                             RESULTS : in RESULTS_STRING) ;
```

c. Send data through channel:

```
function IPC_ENTRY_CALL (CONTEXT_NAME : in STRING;  
                        CHANNEL_NAME : in STRING;  
                        TIME_LIMIT : in DURATION  
                        := DURATION'LAST;  
                        PARAMS : in PARAMS_STRING)  
return RESULTS_STRING;
```

d. Select channel for IPC: procedure IPC_SELECT is named but not yet defined in this package.

Facilities to create channels are not provided in this package; however, Intermetrics has indicated verbally that channel creation will be provided. [TT83]

5.4 KAPSE Database Interfaces

5.4.1 Sufficient Interfaces. The KAPSE Database Services interfaces are sufficiently defined for the AIM implementation in the package TEXT_IO: ([DOD82] 14.3.10)

1. Open/Close a database file

Package TEXT_IO contains Open and Close procedures :
([DOD82] p 14-2,3)

```
procedure OPEN (FILE : in out FILE_TYPE ;  
                MODE : in FILE_MODE := OUT_FILE ;  
                NAME : in STRING ;  
                FORM : in STRING) ;
```

```
procedure CLOSE (FILE : in out FILE_TYPE) ;
```

2. Read/write data to a database file

The TEXT_IO package defined in [DOD82] contains PUT and GET procedures for file I/O which support variable length strings: ([DOD82] p 14-19) String I/O is accomplished by calls to PUT and GET single characters for the length of the string.

```
procedure PUT (FILE : in FILE_TYPE; ITEM : in STRING) ;
```

```
procedure GET (FILE : in FILE_TYPE; ITEM : out STRING) ;
```

3. Create database files

Procedure CREATE in TEXT_IO allows the AIM to create database file objects. ([DOD82] p 14-3)

```
procedure CREATE (FILE : in out FILE_TYPE ;  
                 MODE : in FILE_MODE := DEFAULT_MODE ;  
                 NAME : in STRING := "" ;  
                 FORM : in STRING := "") ;
```

4. Destroy database files

Procedure DELETE in package TEXT_IO is sufficient for file deletion. ([DOD82] p 14-4)


```
procedure DELETE (FILE : in out FILE_TYPE) ;
```

5.5 Miscellaneous Interfaces

5.5.1 Sufficient Interfaces.

1. Date information.

Package CALENDAR defined in ([DOD82] p 9-11) specifies several functions and one general procedure for date retrieval:

```
function YEAR    (DATE : TIME) return YEAR_NUMBER ;  
function MONTH   (DATE : TIME) return MONTH_NUMBER ;  
function DAY     (DATE : TIME) return DAY_NUMBER ;  
  
procedure SPLIT (DATE      : in TIME ;  
                 YEAR      : out YEAR_NUMBER ;  
                 MONTH     : out MONTH_NUMBER ;  
                 DAY       : out DAY_NUMBER ;  
                 SECONDS   : out DAY_DURATION) ;
```

5.5.2 Insufficient Interfaces.

1. Call-tree information.

Call-tree information services could not be found in the AIE documents. The ability to access call-tree information is implied by procedures such as EXIT_PROGRAM, which can wait for sub-contexts to complete. However, a procedure which explicitly allows the user to query the call-tree is not provided. ([INT82] p 106)

SECTION 6

STANDARD INTERFACE SET

6.1 General

The SIS package specifications evaluated in this Interface Report are currently under development by the KIT and KITIA. The SIS document analyzed here is only a preliminary draft, dated February 1983. Subsequent versions of the SIS will be evaluated in subsequent interface reports. The package specifications currently lack semantic explanations, which necessitates some speculation. With this in mind, this section examines the SIS interface provisions which do exist, outlining the AIM requirements which are met and identifying those which are insufficient or missing.

6.2 Background

The Standard Interface Set (SIS), described in [SIS83], is a result of an effort by the KAPSE Interface Team (KIT) and the SIS Drafters group to provide APSE users with a transportable interface to any KAPSE. Ada tools which use only SIS packages should be portable to any APSE which has an implementation of the SIS. The areas the SIS addresses are:

1. Input/Output
2. Database management
3. Process management
4. Utilities

6.3 Terminal Communication Services Interfaces

6.3.1 Sufficient Interfaces. The SIS defines several I/O packages which support primitive terminal functions required by the AIM as follows:

1. Enable/disable echo.

The SIS defines procedure SET_ECHO in package INTERACTIVE_IO which allows echo to be turned on or off, and function ECHO which queries the current status of echoing: ([SIS83] p 14)

```
procedure SET_ECHO (FILE: in FILE_TYPE;
                   TO: in boolean := true) ;

function ECHO      (FILE: in FILE_TYPE) return boolean ;
```

2. Write variable length strings.

The SIS supports string I/O in package SIS_TEXT_IO, procedure PUT: ([SIS83] p 10)

```
procedure PUT (FILE: in FILE_TYPE; ITEM : in STRING) ;
```

The SIS also supports a READ and WRITE in the generic package SIS_SEQUENTIAL_IO, which perform I/O on the private type ELEMENT_TYPE, potentially a string. ([SIS83] p 4)

3. Screen-oriented facilities.

The SIS provides one procedure to set line and column cursor positioning in package SIS_INTERACTIVE_IO: ([SIS83] p 14)

```
procedure SET_CURSOR (FILE: in FILE_TYPE;
                     TO : in CURSOR_TYPE) ;
```

5.3.2 Insufficient Interfaces. The following AIM terminal services requirements are not described or are yet undeveloped in the SIS documentation:

1. Read single character - unable to determine.

Whether I/O is buffered or not is largely implementation dependent, sometimes contingent upon host machine services. The SIS makes no provision for specifying "no-buffered" I/O modes.

2. Exclusive access to user terminal - unable to determine.

The SIS describes the skeleton of a generic device control package called SIS_DEVICE_CONTROL. Package TERMINAL CONTROL is defined but not yet elaborated. It is possible that this

package may provide some control over user terminal access.

3. Write exactly as internally represented - unable to determine.

This requirement, like "read single character" above, is largely implementation dependent. The SIS does not describe a "passthrough" mode (as VMS defines) to write byte strings with no system translation.

4. Terminal identification - unable to determine.

The terminal capabilities in package `TERMINAL_CONTROL` ([SIS83] p 15) have not yet been defined. It is possible that this package may contain a procedure similar to the AIE's `GET_TERMINAL_INFO` which allows the AIM to query the `TERMINAL_INFO_BLOCK` to obtain terminal identification data.

6.4 APSE Program Control and Communication Interfaces

6.4.1 Sufficient Interfaces. The package `SIS_PROCESS_STRUCTURE` provides program control support consisting of process creation and termination. Processes may be explicitly started using the `SPAWN` procedure. Several options are available for terminating and aborting processes and their spawned processes.

The package `SIS_PROCESS_CONTROL` supports process suspension, resumption, and abortion. Process status queries may be satisfied using the `STATUS` function from this package. The procedures which satisfy the AIM APSE program interfaces are as follows:

1. Initiate APSE program from within another APSE program.

When an Ada program or task is invoked, a process is created to represent the execution of the program. By default, each process is created as a subprocess of its creator ([SIS83] p 25). Procedure `INVOKE` in package `SIS_PROCESS_STRUCTURE` invokes an APSE program and waits for it to complete. Procedure `SPAWN` in the same package also invokes an APSE program, but the calling program continues execution: ([SIS83] p 27)

```

procedure INVOKE (KEY: in KEY_STRING;
                  PARAMS: in PARAMS_STRING;
                  RESULTS: in out RESULTS_STRING;
                  STATUS: out PROCESS_STATUS ;
                  STD IN: in FILE_TYPE:=
                      SIS_TEXT_IO.CURRENT_INPUT;
                  STD OUT: FILE_TYPE:=
                      SIS_TEXT_IO.CURRENT_OUTPUT;
                  STD ERR : in FILE_TYPE:=
                      SIS_INTERACTIVE_IO.CURRENT_ERROR;
                  LOCATION : in PROCESS_LOCATION := 0 ) ;

```

```

procedure SPAWN (KEY: in KEY_STRING;
                 PARAMS: in PARAMS_STRING;
                 STD IN: in FILE_TYPE :=
                     SIS_TEXT_IO.CURRENT_INPUT;
                 STD OUT: in FILE_TYPE :=
                     SIS_TEXT_IO.CURRENT_OUTPUT;
                 STD ERR: in FILE_TYPE :=
                     SIS_INTERACTIVE_IO.CURRENT_ERROR;
                 LOCATION: in PROCESS_LOCATION := 0 ) ;

```

2. Suspend execution.

Package SIS_PROCESS_CONTROL provides a suspension primitive:
([SIS83] p 31)

```

procedure SUSPEND (NAME: in NAME_STRING) ;

```

The process identified by NAME and all of its subprocesses will be suspended from execution.

3. Resume execution.

A process resumption primitive is included in the same package: ([SIS83] p 31)

```

procedure RESUME (NAME : in NAME_STRING) ;

```

The suspended process identified by NAME and all of its suspended subprocesses will be restarted.

4. Abort program.

Package SIS_PROCESS_CONTROL defines process abortion as a primitive: ([SIS83] p 32)

```
procedure ABORT (NAME: in NAME_STRING) ;
```

This procedure aborts any named process. Package SIS_PROCESS_STRUCTURE also defines a procedure to terminate the current running process and all of its subprocesses. The difference in functionality is not clear from the document. ([SIS83] p 28)

```
procedure ESCAPE (RESULTS: in RESULTS_STRING) ;
```

5. Determine program status.

Package SIS_PROCESS_CONTROL contains a procedure which allows the AIM to query the status of programs running under its control: ([SIS83] p 32)

```
function STATUS (NAME : in KEY_STRING)  
    return PROCESS_STATUS ;
```

6.4.2 Insufficient Interfaces. Program control interface issues are intricate and sometimes implementation dependent. The preliminary SIS draft [SIS83] does not elaborate on the semantics of the packages it defines, so these AIM requirements are considered yet unsupported by the SIS:

1. Intercept APSE program's terminal I/O - unable to determine.

The SIS defines the OPEN procedure in package SIS_TEXT_IO to contain a FORM mode parameter: ([SIS83] p 8)

```
procedure OPEN (FILE : in out FILE_TYPE ;  
    MODE : in FILE_MODE ;  
    NAME : in STRING ;  
    FORM : in STRING := "") ;
```

It is possible that the SIS intends to declare a "RESERVE MODE" similar to that found in the AIE, to permit SHARED_STREAM access. It is not clear that even this functionality will be appropriate for AIM requirements (see 4.3.2.1). SHARED_STREAM access requires the user to explicitly open the file, which is not normally done with STD_IN and STD_OUT.

2. Exclusive access to the APSE program's terminal I/O - unable to determine.

The terminal I/O packages are currently undefined in the SIS; this feature may be implementation dependent.

3. Interprogram Communication.

The `SHARED_STREAM` mode described above (if it exists) could provide the AIM with the required interprogram communication functionalities. As stated in the AIE rationale, it is not clear that the AIM will utilize IPC facilities, but for the sake of completeness, a parallel analysis of the SIS IPC is provided below.

The SIS defines package `SIS_PROCESS_COMMUNICATION` which contains minimal provisions for interprocess entry and accept calls. A conforming implementation of the SIS must support up to twenty simultaneous accepting channels from one process. No method for creating these channels is defined. The following procedures form the basis for interprocess communication in the SIS: ([SIS83] p 33)

```
procedure ENTRY_CALL (NAME      : in NAME_STRING;
                      CHANNEL    : in CHANNEL_STRING;
                      PARAMS     : in PARAMS_STRING;
                      RESULTS    : in out RESULTS_STRING;
                      LIMIT      : in CALENDAR.DURATION :=
                                CALENDAR.DURATION'LAST);

procedure ACCEPT_ENTRY (CHANNEL : in CHANNEL_STRING;
                       RESULTS  : in RESULTS_STRING);
```

The `ENTRY_CALL` procedure passes the parameters to the process and suspends itself until an `ACCEPT` is performed on the channel (before time limit expires). Procedure `ACCEPT_ENTRY` waits on an `ENTRY_CALL` to the named channel or until the time limit expires.

6.5 KAPSE Database Interfaces

6.5.1 Sufficient Interfaces. The SIS considers its database representation to be a forest of trees with each tree consisting of a node (a node being either a file or a directory). Nodes may be created, deleted, opened, closed, read from, written to, copied, and queried for status. The following procedures fulfill the database service requirements of the AIM:

1. Open/Close a database file.

Package SIS_TEXT_IO provides procedures to open and close a file. ([SIS83] p 8)

```
procedure OPEN (FILE : in out FILE_TYPE ;  
                MODE : in FILE_MODE ;  
                NAME : in STRING ;  
                FORM : in STRING := "") ;  
  
procedure CLOSE (FILE : in out FILE_TYPE) ;
```

2. Read from/Write to database files.

The packages SIS_TEXT_IO, SIS_DIRECT_IO, and SIS_SEQUENTIAL_IO define PUT and GET procedures for various database file types. Following are the string I/O read and write routines from SIS_TEXT_IO: ([SIS83] p 10)

```
procedure GET (FILE : in FILE_TYPE; ITEM : out STRING) ;  
  
procedure PUT (FILE : in FILE_TYPE; ITEM : in STRING) ;
```

3. Create database files.

The package SIS_TEXT_IO provides a file creation procedure: ([SIS83] p 8)

```
procedure CREATE (FILE : in out FILE_TYPE;  
                 MODE : in FILE_MODE := OUT_FILE;  
                 NAME : in STRING := "";  
                 FORM : in STRING := "");
```

4. Destroy database files.

The same package provides for file deletion: ([SIS83] p 8)

```
procedure DELETE (FILE : in out FILE_TYPE);
```

6.6 Miscellaneous Interfaces

1. Date Information.

The SIS contains a package called PACKAGE_CALENDAR which handles current date retrieval ([SIS83] p 36). The procedures in this package are yet undefined.

2. Call-tree information.

Definitions for Call-Tree information retrieval could not be found in the SIS draft. However, this capability is implied through procedures such as SUSPEND which must have access to call-tree information to suspend subprocesses of a named process. The information must exist; an access method has not yet been defined.

SECTION 7

AIM PORTABILITY ISSUES

7.1 General Portability Issues

The transportability of any APSE program, and specifically the AIM, will depend largely on consistency among APSEs. The AIM design emphasizes portability; however, there are functions that by the nature of existing APSEs must be environment dependent. Separate packages must be written for both the ALS and the AIE to isolate and accommodate these dependencies. The SIS package currently under development by the KIT and KITIA (see previous section) would provide a standardized interface to environment dependent functions, greatly enhancing program portability.

7.2 AIM Environment Dependencies

Most functional areas of the AIM are environment independent, although differences in design approach between the ALS and AIE force the AIM to utilize environment dependent features of each APSE for some capabilities. The database services and text I/O tend to be similar between the ALS and AIE, however, interprocess communication and KAPSE terminal services are often quite different in design. The SIS package augments features of both APSEs, and adds design differences of its own.

7.3 AIM Environment Dependent Areas

Below is a comparison of the ALS and AIE environment dependent interfaces relevant to the AIM. SIS procedures are also listed to illustrate the merit of the standard interfaces in resolving APSE transportability problems. For detailed functional analysis of each interface, see the AIE, ALS, or SIS sections of this document.

7.3.1 Terminal Communication Services.

1. Enable/Disable Character Echo.

ALS:

Not supported.

AIE:

```
procedure SET_ECHO (INPUT : in FILE_TYPE ;
                    OUTPUT: in FILE_TYPE);
procedure NO_ECHO  (INPUT : in FILE_TYPE);
```

SIS:

```
procedure SET_ECHO (FILE : in FILE_TYPE;
                    TO : in boolean:= true);
```

2. Exclusive access to the user terminal.

ALS:

Not supported.

AIE:

Intermetrics has stated that this interface will exist, but the method is not yet documented.

SIS:

The SIS describes package SIS_DEVICE_CONTROL which contains package TERMINAL CONTROL. This currently unelaborated package potentially may support this interface.

3. Write to terminal exactly as represented.

ALS:

The ALS does not directly support this interface. The AIM must bypass the KAPSE to use VMS services, which allow the AIM to specify terminal characteristics in the device driver. The TTSM_PASSALL characteristic may be set to ensure that all terminal I/O is binary and no character interpretation is performed.

AIE:

Intermetrics indicated verbally that character strings may be sent to the terminal with no translation. Specific methods

have not yet been defined.

SIS:

Not supported.

4. Terminal Identification.

ALS:

Not supported.

AIE:

```
procedure GET_TERMINAL_INFO (TERM : in integer;
                             INFO : out TERMINAL_INFO_BLOCK);
```

SIS:

Package TERMINAL_CONTROL potentially may support this interface, however, the interface is currently undefined.

5. Screen-oriented Facilities.

ALS:

Not provided.

AIE:

```
procedure SET_LINE (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_COL  (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_OFFSET (FILE : in FILE_TYPE; TO : in COUNT);
```

SIS:

```
procedure SET_CURSOR (FILE : in FILE_TYPE;
                      TO   : in CURSOR_TYPE);
```

7.3.2 APSE Program Interfaces.

1. Initiate program.

ALS:

```

procedure CALL_WAIT (PROGRAM_NAME : in
                     KAPSE_DEFS.SHORT_ID_STRING;
                     PROGRAM_FILE : in
                     KAPSE_DEFS.NODE_NAME;
                     PARAMETER_LIST : in
                     PROG_DEFS.PARM_LIST_REC;
                     STDIN_FILE : in KAPSE_DEFS.NODE_NAME;
                     STDOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                     MSGOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                     PROGRAM_STATUS : in out
                     PROG_DEFS.CALL_STATUS_REC);

procedure CALL_NO_WAIT (PROGRAM_NAME : in
                       KAPSE_DEFS.SHORT_ID_STRING;
                       PROGRAM_FILE : in KAPSE_DEFS.NODE_NAME;
                       PARAMETER_LIST : in
                       PROG_DEFS.PARM_LIST_REC;
                       STDIN_FILE : in KAPSE_DEFS.NODE_NAME;
                       STDOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                       MSGOUT_FILE : in KAPSE_DEFS.NODE_NAME;
                       PROGRAM_STATUS : in out
                       PROG_DEFS.CALL_STATUS_REC);

```

AIE:

```

function CALL_PROGRAM (PROGRAM_PATH : in STRING;
                      PARAMETERS : in PARAMS_STRING;
                      CONTEXT_NAME : in STRING :=
                      ".SUB_CONTEXT";
                      STD_IN : in TEXT_IO.FILE_TYPE :=
                      CURRENT_INPUT;
                      STD_OUT : in TEXT_IO.FILE_TYPE :=
                      CURRENT_OUTPUT)
return RESULTS_STRING;

procedure INITIATE_PROGRAM (PROGRAM_PATH : in STRING;
                          PARAMETERS : in PARAMS_STRING;
                          CONTEXT_NAME : in STRING;
                          STD_IN : in TEXT_IO.FILE_TYPE;
                          STD_OUT : in TEXT_IO.FILE_TYPE);

```

SIS:

```

procedure INVOKE (KEY : in KEY_STRING;
                  PARAMS : in PARAMS_STRING;
                  RESULTS : in out RESULTS_STRING;
                  STATUS : out PROCESS_STATUS;
                  STD_IN : in FILE_TYPE :=
                      SIS_TEXT_IO.CURRENT_INPUT;
                  STD_OUT : in FILE_TYPE :=
                      SIS_TEXT_IO.CURRENT_OUTPUT;
                  STD_ERR : in FILE_TYPE :=
                      SIS_INTERACTIVE_IO.CURRENT_ERROR;
                  LOCATION : in PROCESS_LOCATION := 0);

```

```

procedure SPAWN (KEY : in KEY_STRING;
                 PARAMS : in PARAMS_STRING;
                 STD_IN : in FILE_TYPE :=
                     SIS_TEXT_IO.CURRENT_INPUT;
                 STD_OUT : in FILE_TYPE :=
                     SIS_TEXT_IO.CURRENT_OUTPUT;
                 STD_ERR : in FILE_TYPE :=
                     SIS_INTERACTIVE_IO.CURRENT_ERROR;
                 LOCATION : in PROCESS_LOCATION := 0);

```

2. Suspend program.

ALS:

```

procedure REQ_SUSPENSION (PROG_NAME : in
                          STRING_UTIL.VAR_STRING_REC;
                          REQUEST_STATUS : out
                          PROG_DEFS.PROCAL_STATUS_ENU);

```

AIE:

```

procedure SUSPEND_PROGRAM (CONTEXT_NAME : in STRING);

```

SIS:

```

procedure SUSPEND (NAME : in NAME_STRING);

```

3. Resume program.

ALS:

```
procedure REQ_RESUMPTION (PROG_NAME : in
                           STRING_UTIL.VAR_STRING_REC;
                           REQUEST_STATUS : out
                           PROG_DEFS.PROCAL_STATUS.ENU);
```

AIE:

```
procedure RESUME_PROGRAM (CONTEXT_NAME : in STRING);
```

SIS:

```
procedure RESUME (NAME : in NAME_STRING);
```

4. Abort program.

ALS:

```
procedure REQ_ABORTION (PROG_NAME : in
                        STRING_UTIL.VAR_STRING_REC;
                        REQUEST_STATUS : out
                        PROG_DEFS.PROCAL_STATUS.ENU);
```

AIE:

```
procedure EXIT_PROGRAM (RESULTS : in RESULTS_STRING;
                        ABORT_SUB_CONTEXTS : in boolean
                        := false);
```

SIS:

```
procedure ABORT (NAME : in NAME_STRING);
```

5. Determine program status.

ALS:

```
procedure REQ_STATUS (PROG_NAME : in
                      STRING_UTIL.VAR_STRING_REC;
                      PROGRAM_STATUS : out PROGRAM_INFO_REC);
```

AIE:

Implied but not supported.

SIS:

```
function STATUS (NAME : in NAME_STRING) return PROCESS_STATUS;
```

6. Interprocess communication.

ALS:

Not supported.

AIE:

```
function IPC_ACCEPT (CHANNEL_NAME : in STRING;
                     TIME_LIMIT : in DURATION :=
                     DURATION'LAST) return
                     PARAMS_STRING;
```

```
procedure IPC_END_RENDEZVOUS (CHANNEL_NAME : in STRING;
                             RESULTS : in RESULTS_STRING);
```

```
function IPC_ENTRY_CALL (CONTEXT_NAME : in STRING;
                         CHANNEL_NAME : in STRING;
                         TIME_LIMIT : in DURATION :=
                         DURATION'LAST) return
                         RESULTS_STRING;
```

SIS:

```
procedure ENTRY_CALL (NAME : in NAME_STRING;
                     CHANNEL : in CHANNEL_STRING;
                     PARAMS : in PARAMS_STRING;
                     RESULTS : in out RESULTS_STRING;
                     LIMIT : in CALENDAR.DURATION :=
                     CALENDAR.DURATION'LAST);
```

```
procedure ACCEPT_ENTRY (CHANNEL : in CHANNEL_STRING;
                       RESULTS : in RESULTS_STRING);
```

7. Intercept APSE program's terminal I/O.

ALS:

Not provided. The AIM may bypass the KAPSE and access VMS

mailbox facilities to create channels.

AIE:

The following is a procedure call which may provide this interface:

```
OPEN (FILENAME,IN_FILE, "RESERVE_MODE=>SHARED_STREAM");
```

SIS:

The following procedure call may provide this capability in the SIS:

```
OPEN (FILE,MODE,NAME,FORM : in STRING
      := "RESERVE_MODE");
```

7.3.3 Database interfaces.

1. Create a database file.

ALS:

```
procedure MAKE_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;
                     NAME : in STRING_UTIL.VAR_STRING_REC;
                     MODE : in IO_DEFS.IO_MODE_ENU;
                     FILE_CLASS : in IO_DEFS.FILE_CLASS_ENU;
                     RECORD_LENGTH : in IO_DEFS.DATA_INDEX_INT;
                     RESULT : out IO_DEFS.IO_RESULT_ENU;
                     RESULT_STRING : in out
                     STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
procedure CREATE (FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := DEFAULT_MODE;
                 NAME : in STRING := "";
                 FORM : in STRING := "");
```

SIS:

```
procedure CREATE (FILE : in out FILE_TYPE;  
                  MODE : in FILE_MODE := DEFAULT_MODE;  
                  NAME : in STRING := "";  
                  FORM : in STRING := "");
```

2. Destroy database files.

ALS:

```
procedure DELETE_FILE (STREAM : in IO_DEFS.STREAM_ID_PRIV;  
                      RESULT : out IO_DEFS.IO_RESULT_ENU;  
                      RESULT_STRING : in out  
                      STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
procedure DELETE (FILE : in out FILE_TYPE);
```

SIS:

```
procedure DELETE (FILE : in out FILE_TYPE);
```

3. Open/Close a database file.

ALS:

```
procedure OPEN_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;  
                   NAME : in STRING_UTIL.VAR_STRING_REC;  
                   MODE : in IO_DEFS.IO_MODE_ENU;  
                   FILE_CLASS : out IO_DEFS.FILE_CLASS_ENU;  
                   RECORD_FORMAT : out  
                   IO_DEFS.RECORD_FORMAT_ENU;  
                   RECORD_LENGTH : out  
                   IO_DEFS.DATA_INDEX_INT;  
                   RESULT : out IO_DEFS.IO_RESULT_ENU;  
                   RESULT_STRING : in out  
                   STRING_UTIL.VAR_STRING_REC);
```

```
procedure CLOSE_FILE (STREAM : out IO_DEFS.STREAM_ID_PRIV;  
                    RESULT : out IO_DEFS.IO_RESULT_ENU;  
                    RESULT_STRING : in out  
                    STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
procedure OPEN (FILE : in out FILE_TYPE;  
                MODE : in FILE_MODE;  
                NAME : in STRING;  
                FORM : in STRING := "");  
  
procedure CLOSE (FILE : in out FILE_TYPE);
```

SIS:

```
procedure OPEN (FILE : in out FILE_TYPE;  
                MODE : in FILE_TYPE;  
                NAME : in STRING;  
                FORM : in STRING := "");  
  
procedure CLOSE (FILE : in out FILE_TYPE);
```

4. Read from a database file.

ALS:

```
procedure READ_FILE (STREAM : in IO_DEFS.STREAM ID PRIV;  
                    BUFFER : in KAPSE_DEFS.ADDRESS.INT;  
                    LENGTH : in IO_DEFS.DATA INDEX INT;  
                    COUNT : out IO_DEFS.DATA LENGTH INT;  
                    RESULT : out IO_DEFS.IO_RESULT_ENU;  
                    RESULT_STRING : in out  
                        STRING_UTIL.VAR_STRING_REC);
```

AIE:

```
procedure GET (FILE : in FILE_TYPE; ITEM : out STRING);
```

SIS:

```
procedure GET (FILE : in FILE_TYPE; ITEM : out STRING);
```

5. Write to a database file.

ALS:

```

procedure WRITE_FILE (STREAM : in IO_DEFS.STREAM_ID PRIV;
                      BUFFER : in KAPSE_DEFS.ADDRESS_INT;
                      LENGTH : in IO_DEFS.DATA_INDEX_INT;
                      COUNT  : out IO_DEFS.DATA_LENGTH_INT;
                      RESULT  : out IO_DEFS.IO_RESULT_ENU;
                      RESULT_STRING : in out
                        STRING_UTIL.VAR_STRING_REC);

```

AIE:

```

procedure PUT (FILE : in FILE_TYPE; ITEM : in STRING);

```

SIS:

```

procedure PUT (FILE : in FILE_TYPE; ITEM : in STRING);

```

7.3.4 Miscellaneous interfaces.

1. Date information.

ALS:

```

procedure GET_TIME_DATE (TIME_INFO : out
                        MISC_DEFS.TIME_INFO_REC;
                        DATE_INFO : out
                        MISC_DEFS.DATE_INFO_REC);

```

AIE:

```

function YEAR (DATE : TIME) return YEAR_NUMBER;

function MONTH (DATE : TIME) return MONTH_NUMBER;

function DAY (DATE : TIME) return DAY_NUMBER;

```

SIS:

PACKAGE CALENDAR is defined but not elaborated.

2. Call-tree information.

ALS:

```
procedure REQ_STATUS (PROG_NAME : in STRING_UTIL.VAR STRING_REC;
                     PROGRAM_STATUS : out PROGRAM_INFO_REC);
```

AIE:

Not supported.

SIS:

Not supported.

7.4 KAPSE Document Quality

Since the size of the KAPSE design itself is so large, it is not surprising that there were some areas of design documentation for both KAPSEs that were not sufficiently complete to resolve some issues raised by AIM design. A primary goal of the AIM project is to identify the issues inherent in transporting a tool between the AIE and the ALS. Document completeness and consistency is a significant part of this study. Below are the areas which were found to be weak in both the AIE and ALS documents [SOF82] and [INT82]:

1. Terminal control and communication

- a. Is it possible to read or write a single character?
- b. Can exclusive access to the user terminal be granted to the AIM?
- c. Is it possible to implement a terminal identification scheme?
- d. May a character string be written to a terminal exactly as represented internally (with no characters added or removed)?

2. Process control and communication

- a. Can the AIM intercept APSE program's terminal I/O?
- b. Can the AIM obtain exclusive access to the APSE program's terminal I/O?

7.4.1 Document Deficiencies. Both the AIE documentation [INT82] and the ALS documentation [SOF82] could be improved in the areas of terminal communication and APSE program interfaces. Many of the questions enumerated above were answered by ARPANET response and telephone or verbal communication from SofTech and Intermetrics.

The major problem with both sets of documentation is ambiguity spawned from incompleteness. Ideally, the documents define what the systems support and only that. Some of the questions generated from AIM interface issues were not addressed by the documentation at all. An important question is raised: do the ALS and AIE not support what is not explicitly defined? Or do they only support what they do define? Semantic explanation and examples would help to alleviate this confusion. Speculation upon the extent of document completeness and validity can lead to design and implementation problems.

SECTION 8

USER INTERFACES

8.1 Constraints on User Programs

Although the AIM is intended to be transparent to APSE programs, it is possible that APSE programs may be required to follow some guidelines in order to interface with the AIM, especially in the area of terminal I/O. This section describes some restrictions that might be imposed upon programs which are intended to execute under the AIM.

8.1.1 APSE Program I/O. If the appropriate KAPSE services are provided by the AIE and ALS, the AIM mechanism required to intercept program I/O will not impact APSE program design. APSE programs should need no special I/O file parameters to function in the AIM. APSE program terminal-destined I/O must be accomplished through the STANDARD IN and STANDARD OUT files; an APSE program which accesses the terminal in any other way might not be AIM-compatible. APSE programs should generate only the standard printable ASCII characters. If an APSE program generates characters outside the ASCII printable range, the AIM may interpret the characters as AIM control sequences with unexpected results.

In order for the AIM to obtain exclusive access to an APSE program's terminal input and output, the user program may use only STANDARD IN and STANDARD OUT for terminal I/O. There may be no other files associated with the terminal. Multiple terminals are not currently supported, since neither the ALS or AIE defines a method of associating a terminal with a unique name accessible from within the APSE program. Terminal I/O is accomplished through these standard files which are connected to the terminal. No logical name translation is provided to access a unique device. (The ALS, however, allows the user to bypass the KAPSE to use VMS services to perform logical name translation. A program which bypasses the KAPSE is by definition not an APSE program.)

An APSE program is by definition a program which uses only KAPSE services. Therefore, a program which bypasses KAPSE services for any reason is not an APSE program and therefore might not be AIM compatible. For example, the ALS permits the following:

```
Open ("<<VMS>>TT:")
```

This statement causes an implicit ALS ESCAPE to the underlying host operating system (VMS, in this case). Host services are used to open the terminal file. This statement is not portable and is considered erroneous.

8.1.2 MASTER IN, MASTER OUT, and MESSAGE OUT. The ALS defines two extraneous files for I/O besides STANDARD IN and STANDARD OUT, called MSTR IN and MSTR OUT. These two extra files are always associated with the terminal. The purpose of these files was not clear from the documents. Verbal communications [RT83A] indicated that MSTR IN and MSTR OUT are provided to allow batch streams to send status messages to the terminal, such as "please mount tape". There is no way to disassociate MSTR IN and MSTR OUT from the terminal; these messages will always be directed to the screen. This could disrupt the user of an interactive program (such as the AIM or a text editor) if a batch stream sends a message which demands a response.

The AIM restricts APSE programs to use only a single pair of files for terminal I/O, specifically STANDARD IN and STANDARD OUT. Therefore, using MSTR IN and MSTR OUT by definition creates an APSE program which might not run correctly under the AIM. The ALS additionally defines MSG OUT, a file which is always associated with the terminal (presumably intended for system messages, [RT83A]). Use of this file will also create problems when executing under the AIM.

8.2 Command Language Processor Constraints

The KAPSE Command Language Processor is treated like any other APSE program which may be invoked from the AIM. Therefore, there are no user interface problems which arise.

SECTION 9

KAPSE ISSUES

9.1 General

This section describes features provided in both the ALS and the AIE KAPSEs which may adversely affect the function of the AIM and APSE programs invoked from the AIM.

9.2 Bypassing KAPSE Services for Program Control

Both the ALS and the AIE provide the user with the capability to suspend program and terminal I/O from the terminal: the ALS "break-in" facility and the AIE "scroll mode control". This allows the user to suspend the AIM itself. The user has control of the terminal and may randomly change screen data without AIM supervision. When AIM execution is resumed, the user may become confused because the AIM assumes that mappings between AIM images and the actual display are intact, when in reality they have been changed.

9.2.1 ALS "Break-In" facility. The ALS allows a user to type <cntl>-C to receive control of any currently running job. This gives the user control of the screen and terminal-directed I/O. Since the AIM is an APSE program, it may be suspended by the "break-in" facility, which returns control of the user's terminal I/O to the KAPSE. If the user changes the information on the screen and then resumes AIM execution, the results are unpredictable. The AIM makes certain assumptions about the screen and mappings among images, windows, and viewports, and if the user moves or deletes screen information, the display will not correctly reflect AIM mappings. Therefore, the use of the ALS "break-in" key is potentially hazardous to the AIM user.

If the "break-in" facility is suspendable, however, these problems could be alleviated because the AIM would then control the break-in. The break-in facility would even become useful for the AIM itself to control programs and terminal I/O. Conversely, the facility is designed as an "emergency" mechanism to provide the user absolute control, and implementing it as a program may defeat its purpose.

Another possibility is that the KAPSE might reinstate the screen status automatically upon program resumption. This would ensure that the AIM screen mappings would remain valid even if the AIM is suspended.

The AIM cannot automatically refresh the screen upon resumption because it has no knowledge of having been suspended. The user must choose to restore the screen display arbitrarily. The AIM itself will provide a screen refresh function invoked by a special key sequence. If the AIM is suspended for some reason (intentional or unintentional) and the screen data is modified, this function will reinstate the correct mappings between the internal AIM images and the screen.

9.2.2 AIE "Scroll Mode Control". The AIE extends the ALS "break-in" facility to include terminal I/O functions ([INT82] p 114). The user types a <CNTL>-S to stop terminal output and enter scroll control mode. This mode is intended to provide a "cache" of output which the user may have lost from scrolling or printer malfunction. Once in scroll control mode, the user has control of terminal I/O and may scroll the screen or perform simple editing functions through a "terminal handler". The user may also interrupt program execution. All terminal input and output is stored in temporary files for historical purposes.

If a user invokes scroll control mode while under the AIM, the consequences are rather unpredictable. It is not clear if all running programs are automatically suspended, or if execution continues. Either result could adversely affect the function of the AIM.

9.3 Broadcast Messages

The AIE enables the AIM to obtain exclusive access of the user terminal I/O. The ALS does not provide a mechanism for granting the AIM exclusive access to the user terminal. The absence of exclusive access enables the host operating system to generate system "broadcast" messages which may overwrite portions of the screen. The AIM should allow system messages to be generated, but these messages may disturb the progress of an APSE programming session. The user's recourse is to reinstate the screen with the AIM refresh function described above.

9.4 Bypassing KAPSE Services for Terminal Control

The ALS permits the user to perform an implicit escape to the underlying host operating system for some terminal control functions (such as opening the terminal file, see para 7.1.1). The use of this feature may alter the appearance of the screen to incorrectly reflect AIM mappings. A program which bypasses KAPSE services in this manner

is not a true APSE program and is considered erroneous.

9.5 Programs targeted for specific terminals

APSE programs which take advantage of specific host capabilities might not interface with the AIM. For example, the ALS imports the VAX EDT editor rather than implementing another one which uses only KAPSE services. This implies that EDT may not work with the AIM, since all APSE programs running under the AIM must only use KAPSE services. Attempting to invoke EDT from the AIM will make terminal I/O unpredictable, since both EDT and the AIM will assume that they each have complete control of the terminal. Obviously, this may lead to control conflicts and undesirable results.

APPENDIX A

AIM INTERFACES SUMMARY

A.1 Interface Comparison

KAPSE Terminal Services

AIM Interface Requirements	ALS	AIE	SIS
Read single character	No [E]	No [D]	No?
Enable/Disable echo	No	Yes [D]	Yes
Exclusive access	No	Yes	?
Write variable length strings	Yes	Yes	Yes
Write exactly as internally represented	No [E]	Yes	?
Terminal identification	No	Yes [A]	No?
Screen-oriented facilities	No	Yes	Yes

Program Control Interfaces

AIM Interface Requirements	ALS	AIE	SIS
Initiate APSE programs from within other APSE programs	Yes	Yes	Yes
- suspend	Yes	Yes	Yes
- resume	Yes	Yes	Yes
- abort	Yes	Yes	Yes
Determine program status	Yes	No	Yes
Intercept APSE program's terminal I/O	?	?	?
Exclusive access to the APSE program's terminal I/O	No [E]	? [D]	?
Interprocess Communication	No	Yes	Yes

Yes => Support provided

No => No support provided

? => Could not determine if support is provided

KAPSF Database Services Requirements

AIM Interface Requirements	ALS	AIE	SIS
Read/write data to a DB file	Yes	Yes	Yes
Create database files	Yes	Yes	Yes
Destroy database files	Yes	Yes	Yes

Miscellaneous Services

Obtain current date	Yes	Yes	Yes
Call-tree information	Yes	No?	No?

Yes => Support provided

No => No support provided

? => Could not determine if support is provided

References:

- [A] Intermetrics Inc., "Draft IR-678-2 Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5," Wakefield, MA, November 1982. p 65.
- [B] ibid, pp 60-62.
- [C] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, June 1981.
- [D] ARPANET response from Intermetrics Inc.
- [E] ARPANET response from SofTech Inc.

A.2 Interface Summary

AIM Interface Area	ALS	AIE	SIS
Terminal communications	*	**	**
APSE programs	**	**	**
Database services	***	***	***
Miscellaneous services	***	**	**

Key :

*** - Satisfactory
** - Minimally deficient
* - Quite deficient

APPENDIX B

ARPANET COMMUNICATIONS

Confusion about document content spawned the following question and answer exchange between Texas Instruments and the APSE contractors Intermetrics (AIE) and SofTech (ALS). Most of this information pertains directly to KAPSE interfaces, so it is included in transcribed form. (Answers are dated 8 Nov 82 for AIE, 12 Nov 82 for ALS.)

1. Question:

Within the KAPSE is there a facility for directly referencing an interactive device? (ie. can character sequences be sent to and received from the device without any translation?)

AIE: The initial KAPSE/Tool interfaces include no mechanism for direct reference to an interactive device. Instead, full-screen terminals are made to look like a text file with random access to line and column (see below).

ALS: In the ALS, interactive devices can be referenced in two ways :

a. Explicitly open the "file" named "<<VMS>>TT:", where TT: is the name that VMS assigns to the device, in this case the terminal.

b. If you want the terminal that the user is connected to, use the predefined and preopened "files" named .MSTRIN (keyboard) and .MSTROUT (terminal display device).

Once open, you will be able to use basic io.read file and basic io.write file to pass byte strings to and from the DEVICE DRIVER. The ALS KAPSE will not do any translation of the bytestrings. HOWEVER, you will have to get by the VMS device driver. This could be the subject of a VMS experiment. The ALS KAPSE does not support any official way of opening a device in "raw" mode. If you can do it by passing bytestrings to the opened devices, then it can be done, otherwise not. I do not know the nature of the character translation performed by the VMS device driver.

2. Question:

Does the KAPSE support any functionality for interactive devices other than teleprinters? Are there any multidimensional capabilities, for example, cursor positioning?

AIE: The KAPSE supports x-y cursor positioning using the primitives of the package SIMPLE_OBJECTS.TEXT_ACCESS (see KAPSE B5 IR-678-1, p. 24), SET_LINE, SET_COL.

ALS: Unless it can be done by passing a byte string, there is no explicit x-y cursor positioning supported by the ALS KAPSE. The notion of a two dimensional display is not supported by the KAPSE. However, the Ada TEXT_IO package should work for CRT based terminals.

3. Question:

Are I/O operations to interactive devices buffered? Must NEW_LINE or PUT_LINE be called before the text is actually sent to the device?

AIE: It is our intention that I/O may be buffered. Probably an end-of-line will cause flushing, but in any case, requesting input from a file which is echoing on the output file (see B5 p. 65, package INTERACTIVE_IO), will cause a flush. Input is buffered up so that local line-editing may be performed before the characters are received as part of the text input file. The initial KAPSE will probably always buffer up input until an ENTER/Carriage Return key is depressed. Eventually, using the SET_INPUT_INFO call of Package INTERACTIVE_IO, more control will be available.

ALS: Keyboard input is buffered by VMS which does the line editing. In general, the KAPSE sees no keyboard input until the line is sent by use of the return key. The exceptions to this are some of the special control operations like control-C and control-Y used for interruption; but these are very special-purpose operations. For most of the standard DEC terminals, the CPU sees each keystroke. I believe that the device driver performs the buffering, not the hardware. The ALS KAPSE does no input buffering itself.

For output, buffering is performed when using Ada TEXT_IO. A new_line or put_line is necessary to obtain the transmission of the characters buffered. Characters are also transmitted when the line length is exceeded. Presumably, the length could be set to 0 or 1, but this would cause the insertion of the line mark after each character. Basic_io.write_file performs no buffering. Every call to this service will result in transmission to the device driver.

4. Question:

Can two or more logical devices have concurrent access rights to an interactive device? (Two "internal files" referencing the users terminal.)

AIE: It will probably be undefined what happens when two programs/tasks try to read from the same terminal input stream (and hence "erroneous" if not an explicit exception). To accomplish your task, I would recommend that your virtual terminal manager be the only process with the terminal input/terminal output open, and that all other processes do interactive I/O by using pipes to the virtual terminal manager. The KAPSE allows multiple (Ada) tasks within the same program to each be doing synchronous pipe I/O, with only the particular task suspended which is actually waiting for input.

Pipe I/O is accomplished using normal TEXT IO, but with the pipe/file opened in "SHARED STREAM" mode (see B5 pp. 40, 41 for explanation of Shared Stream read/write, and p. 25 for explanation of use of FORM string for RESERVE_MODE specification).

ALS: If the device is a terminal, VMS will allow concurrent read and write access by multiple "internal files".

5. Question:

Does the AIE work with 3270-compatible devices?

AIE: The AIE will support 3270 compatible terminals, but will not initially support the field protect/field read features in a way that is useful to the application programmer. Instead, the terminals will be made to look as much like a full-screen ASCII terminal with cursor addressing.

ALS: N/A

APPENDIX C

GLOSSARY

AIE Ada Integrated Environment

AIM APSE Interactive Monitor

ALS Ada Language System

APSE Ada Programming Support Environment

APSE program
A program that can be executed in the hosting APSE and uses only
KAPSE supplied services to perform its function.

character
A member of a set of elements that is used for the organization,
control, or representation of data.

character echo
The act of re-transmitting a character immediately upon receipt of
it back to the entity that originally transmitted it.

character imaging device
A device that gives a visual representation of data in the form of
graphic symbols using any technology, such as cathode ray tube or
printer.

character stream
An unbounded sequence of ASCII characters.

character string
A bounded sequence of ASCII characters.

database file
A standard file in the APSE database.

display

The area for visual presentation of data on a character imaging device.

display terminal

A data communications device composed of a keyboard and a display screen (usually a cathode ray tube).

EDT

An interactive full-screen editor supported by DEC on the VAX machine.

environment-dependent

Using features which are unique to a specific Ada Program Support Environment (such as ALS or AIE).

erroneous

An Ada program which does not conform to the requirements of an APSE program. The program might execute correctly within an APSE in a given situation, but the program may not be considered entirely reliable. An APSE program must use only KAPSE services; any other services (such as host services) result in an erroneous program.

exclusive access

Control of a file (or, the terminal, in this case) which prohibits any other program besides the AIM from writing to the terminal screen.

hardcopy terminal

A data communications device composed of a keyboard and a printer.

host services

Facilities provided by the operating system of the host machine underlying the KAPSE.

image

An analog of the physical display device. The image is the entity that is mapped onto the display. Given a number of user defined images, only one at a time can be mapped onto the display. The rest exist and are updated asynchronously but are not mapped onto the display until the user requests it.

indirect command script

A database file containing commands to a command interpreter (in this case, the AIM command interpreter). The command interpreter reads commands from the indirect command script rather than prompting the user interactively.

interface

The place at which independent systems meet and act on or communicate with each other.

KAPSE

Kernel Ada Programing Support Environment.

keyboard

The physical input device.

KIT

KAPSE Interface Team.

line

A set of adjacent character positions in a visual display that have the same vertical position.

mappings

The relationships managed by the AIM connecting logical representations of windows, images, and viewports to physical representations on a display device.

node

Pertaining to the KAPSE database, either a file or a directory in the tree-structured database.

NOSC

Naval Ocean Systems Center

pad

A file which contains a complete history of window activity that transpires from the beginning of pad mode until it is terminated by the user or the window is destroyed. This includes the input to the APSE program from the user through the keyboard as well as the output to the display from the AIM and any program initiated by the AIM.

pipe

A logical connection between an output file of one program and an input file of another program.

screen

The area for visual presentation of data on any type of character imaging device, including printer and cathode ray tube device.

scroll mode terminal

A display terminal that presents data by moving all the graphic symbols of the screen in one direction to make room for new data.

SIS

Standard Interface Set, the KIT/KITIA effort to standardize certain KAPSE interfaces.

STANDARD IN and STANDARD OUT

Input and output files defined in the package TEXT_IO. For AIM purposes, these must be the only files used for terminal I/O,

task

An Ada program unit that operates in parallel with other program units.

terminal

A data communications device consisting of a keyboard and a character imaging device.

Terminal Capabilities File

A file which describes common terminal functions in terms of device-specific control sequences, for many different terminals.

terminal communication protocols

Sequences of characters in which the relationships between specific characters are given meanings for different types of terminals.

transmit

To send data as a data stream for purposes of information interchange.

user terminal

The terminal with which a user interacts in order to communicate with an APSE program.

VMS

Virtual Memory System, the DEC operating system for the VAX 11-780.

viewport

The portion of the window displayed in the image.

viewport header

A single optional highlighted line located at the top of a viewport.

window

An analog of the APSE program's view of the terminal.

APPENDIX D

REFERENCES

D.1 Government Standards

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [DOD80] United States Department of Defense, "Requirements for Ada Programming Support Environments" ("STONEMAN"), February 1980.
- [DOD82] United States Department of Defense, "Reference Manual for the Ada Programming Language Draft, Revised MIL-STD-1815," July 1982.
- [DOD83] United States Department of Defense, "Reference Manual for the Ada Programming Language Draft, Revised MIL-STD-1815A," January 1983.
- [DID73] Data Item Description, "Informal Technical Information, DI-S-30593," March 73.

D.2 Government Specifications

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the document referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [INT81D] Intermetrics Inc., "Draft IR-679 Computer Program Development Specification for Ada Integrated Environment: MAPSE Command Processor Type B5," Wakefield, MA, March 1981.

- [INT81E] Intermetrics Inc., "Draft IR-680 Computer Program Development Specification for Ada Integrated Environment: MAPSE Generation and Support Type B5," Wakefield, MA, March 1981.
- [INT81F] Intermetrics Inc., "Draft IR-681 Computer Program Development Specification for Ada Integrated Environment: Program Integration Facilities Type B5," Wakefield, MA, March 1981.
- [INT81H] Intermetrics Inc., "Draft IR-683 Computer Program Development Specification for Ada Integrated Environment: MAPSE Text Editor Type B5," Wakefield, MA, March 1981.
- [INT81J] Intermetrics Inc., "IR-684 Ada Integrated Environment (AIE) Design Rationale: Technical Report (Interim)," Wakefield, MA, March 1981.
- [INT82 1] Intermetrics Inc., "IR-678-1 Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5," Wakefield, MA, November 1982.
- [SIS83 1] KAPSE Interface Team (Ada Joint Program Office), "Ada Package Specification for the Standard Interface Set (SIS)" Draft 1, Version 1, February 1983.
- [SOF81A] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, June 1981.
- [SOF81B] SofTech Inc., "Draft Ada Language System KAPSE C5 Specification CR-CP-0059-C83" Waltham, MA, December 1981.
- [SOF81C] SofTech Inc., "Preliminary Draft Ada Language System KAPSE B5 Specification," Waltham, MA, August 1981.
- [SOF82 1] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, August 1982.
- [SOF82A] SofTech Inc., Ada Problem Report #602, Waltham, MA, November 1982.

D.3 Other Government Documents

The following documents of the latest issue per date of this report form a part of this specification.

- [TI82] Texas Instruments, Advanced Computer Systems Laboratory, "Proposal for Development of Ada Software Tools and Interface Standards," Lewisville, TX, February 1982.
- [TI83] Texas Instruments, "AIM Program Performance Specification," (Initial submission) Lewisville, TX, 1 March 1983.

D.4 Special Sources

- [TT83] Verbal communications with Tucker Taft of Intermetrics, Inc., Jan 26, 1983 at the San Diego KIT meeting.
- [TT83A] Verbal communications with Tucker Taft of Intermetrics, Inc., April 21, 1983 at the Willow Grove, PA KIT meeting.
- [RT83] Verbal communications with Rich Thall of SofTech, Inc., Jan 26, 1983 at the San Diego KIT meeting.
- [RT83A] Verbal communications with Rich Thall of SofTech, Inc., April 20, 1983 at the Willow Grove, PA KIT meeting.

D.5 Other Publications

- [AKIN81] Akin, T. Allen, "Virtual Terminal Handler Preliminary Quick Reference," School of Information and Computer Science, Georgia Institute of Technology, April 1981.
- [ANSI73] American National Standards Institute, "American National Standard Graphic Representation of the Control Characters of American National Standard Code for Information Interchange (ANSI Standard X3.32-1973)," July 1973.

- [ANSI77] American National Standards Institute, American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)," June 1977.
- [ANSI79] American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)," July 1979.
- [APSE82] "Working Paper: Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for Standard Interface Specifications," Not Approved, October 1982.
- [COX83] Cox, Fred, "KAPSE Support for Program/Terminal Interaction", Working paper for KITIA/ Working Group 1, February 1983.
- [CSC82A] Computer Sciences Corporation, "Configuration Management System Program Performance Specification (Draft)," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract #N00123-80-D-0364.
- [CSC82B] Computer Sciences Corporation, "Configuration Management System Interim Report on Interface Analysis," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract #N00123-80-D-0364.
- [DEC82] Digital Equipment Corporation, "VAX/VMS I/O User's Guide (Volume 1)", Maynard, Massachusetts, May 1982.
- [DP82] Datapro Reports on Data Communications, vol 2., Sept 1982, "Display Terminals", p C25-10-101
- [FH83] French, Stewart and Harrison, Tim, "The APSE Interactive Monitor" Texas Instruments, Inc., March 1983.
- [FRA] Franck, R., "Design and Implementation of a Virtual Terminal for a Real-time Application System"
- [GREN80] Greninger, Lars and Roberts, Roger, "Considerations for a Local Virtual Terminal Interface," Presented at IEEE Conference, September 1980.

- [ISO642] International Standards Organization, Standard number: ISO DP 6429, "Additional Control Functions for Character Imaging Devices (Draft)," Not approved, April 1982.
- [JOY81] Joy, W. and Horton, M., "TERMCAP," UNIX Programmer's Manual, Seventh Edition, Berkley release 4.1, June 1981.
- [LAN79] Lantz, Keith and Rashid, Richard, "Virtual Terminal Management in a Multiple Process Environment," Proceedings of the Seventh Symposium on Operating Systems (ACM), December 1979.
- [MAG79] Magnee, F., Endrizzi, A., and Day, J, "A Survey of Terminal Protocols," Computer Networks, 1979, pp 299-314.
- [MEY81] Meyrowitz, Norman and Moser, Margaret, "Bruwin: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems," Department of Computer Science, Brown University, December 1981.
- [SCH78] Schicker, P. and Duenki, A., "The Virtual Terminal Definition," Computer Networks, 1978, pp 429-441.
- [STE81] Stenning, Vic, Et Al., "The Ada Environment: A Perspective," Computer, Volume 14, number 6, June 1981, pp 26-34, 36.
- [SUK81] Sukamar, Srinivas and Wiese, John D, "Hardware and Firmware Support for Four Virtual Terminals in One Display Station," Hewlett-Packard Journal, March 1981.
- [TAF82] Taft, S. Tucker, "Portability and Extensibility in the Kernel and Database of a Programming Support Environment," Intermetrics, March 1982.
- [TAJ79] Tajima, Takashi and Katsuyama, Yoshiki, "Layered and Parametric Approach to Terminal Virtualization," Presented at International Conference on Communications, Boston, MA, June 1979.
- [TI81A] Texas Instruments, "Ada Integrated Environment," Lewisville, TX, March 1981. Prepared for Rome Air Development Center (RADC) under DoD Contract F30602-80-C-0293.

- [THA82] Thall, Richard, "The KAPSE for the Ada Language System," SofTech Inc, Proceedings of the Adatec conference on Ada, October 1982.
- [WOL81] Wolfe, Martin I., et al., "The Ada Language System," Computer, Volume 14, number 6, June 1981, pp 37-45.

APPENDIX C



KERNEL ADA* PROGRAMMING SUPPORT
ENVIRONMENT INTERFACE TEAMS' ACTIVITIES

ANTHONY GARGARO

COMPUTER SCIENCES CORPORATION
DEFENSE SYSTEMS DIVISION
TACTICAL SYSTEMS CENTER

ADA-EUROPE/ADATEC CONFERENCE
BRUSSELS, MARCH 1983

* ADA IS A TRADEMARK OF THE U. S. DoD (AJPO).

CSC

BRIEFING OF ACTIVITIES

- BACKGROUND
- ORGANIZATION
- OBJECTIVES
- KAPSE INTERFACES
- STATUS

- NEED FOR SPECIALIZED ENVIRONMENT TO SUPPORT NEW DoD HOL
- CONFERENCES AND WORKSHOPS ON ENVIRONMENT REQUIREMENTS - 1978..1979
- SERIES OF DOCUMENTS CIRCULATED FOR PUBLIC REVIEW
- STONEMAN PUBLISHED - FEBRUARY 1980.

CSC

POST-STONEMAN

- ● PUBLIC REVIEW OF STONEMAN-BASED ADA INTEGRATED ENVIRONMENT (AIE) DESIGNS
- ADA LANGUAGE SYSTEM (ALS) AWARD
- AIE AWARD
- TOOL TRANSPORTABILITY REQUIRED AMONG DIFFERENT APSES
- TRI-SERVICE MEMORANDUM OF AGREEMENT



MEMORANDUM OF AGREEMENT

- FORMALLY ISSUED JANUARY 1982
- ESTABLISHED JOINT SERVICE EVALUATION TEAM
 - NAVY CHAIRED
 - LONG TERM OBJECTIVE: ESTABLISH INTERFACE CONVENTIONS FOR CONVERGENCE OF MULTIPLE EFFORTS IN 1985 TIME-FRAME.

TRI-SERVICE MEMORANDUM OF AGREEMENT

- "WE AGREE WITH THE CONCEPT OF STANDARD TOOL INTERFACES TO THE KAPSE, AND A STANDARD FOR ALL OTHER ASPECTS OF THE KAPSE WHICH ARE VISIBLE TO THE TOOLS."
- "...THE LONG TERM GOAL IS TO ESTABLISH THE NECESSARY INTERFACE CONVENTIONS SO THAT MULTIPLE EFFORTS MAY CONVERGE TO A SINGLE SET OF INTERFACE STANDARDS IN THE 1985 TIME FRAME."



KAPSE INTERFACE TEAM - KIT

- ESTABLISHED LATE 1981
- ESSENTIALLY GOVERNMENT, PRIMARILY NAVY
- CHAIRED BY PATRICIA OBERNDORF, NOSC
- INDUSTRY REPRESENTED BY:
 - COMPUTER SCIENCES CORPORATION
 - INTERMETRICS, INC.
 - SOFTECH, INC.
 - TEXAS INSTRUMENTS
 - TRW

KAPSE INTERFACE TEAM

TRICIA OBERNDORF, NOSC
RICH BALDWIN, CECOM
JINNY CASTOR, AFWAL
BOB CONVERSE, NAVSEA
ED DUDASH, NSWC/DL
RON HOUSE, NUSC
LARRY JOHNSTON, NADC
JACK KRAMER, AJFO
LARRY LINDLEY, NAC
WARREN LOPER, NOSC
H. O. LUBBES, NAVELEX

SHIRLEY PEELE, FCDSSA/DN
LEE PURRIER, FCDSSA/SD
ELIZABETH WALD, NRL
CHUCK WALTRIP, JHAPL
DOUG WHITE, RADC
MARTY WOLFE, CECOM

JOHN FOREMAN, TI
CHARLES FORREST, TRW
HAL HART, TRW
DONN MILTON, CSC
ELDRED NELSON, TRW
TUCKER TAFT, INTERMETRICS
RICH THALL, SOFTECH

CSC

KIT ADDITIONS AND CHANGES

● RECENT ADDITIONS AND CHANGES TO THE KIT:

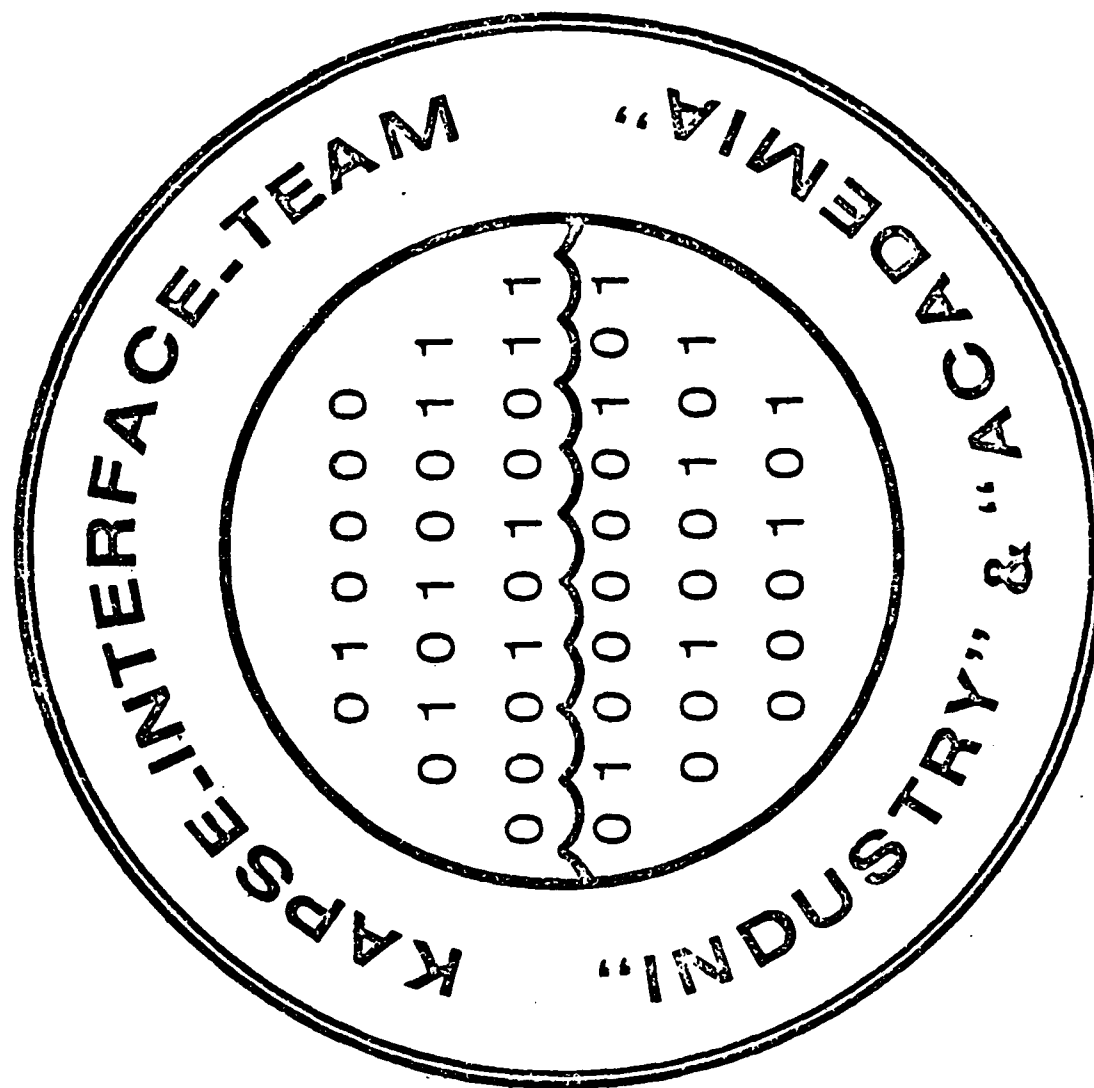
- GIL MYERS, NOSC
- BRIAN SCHAAAR, AJPO
- PHIL MYERS, NAVELEX
- LIZ KEAN, RADC
- JACK FOIDL, TRW
- MITCH BASSMAN, CSC



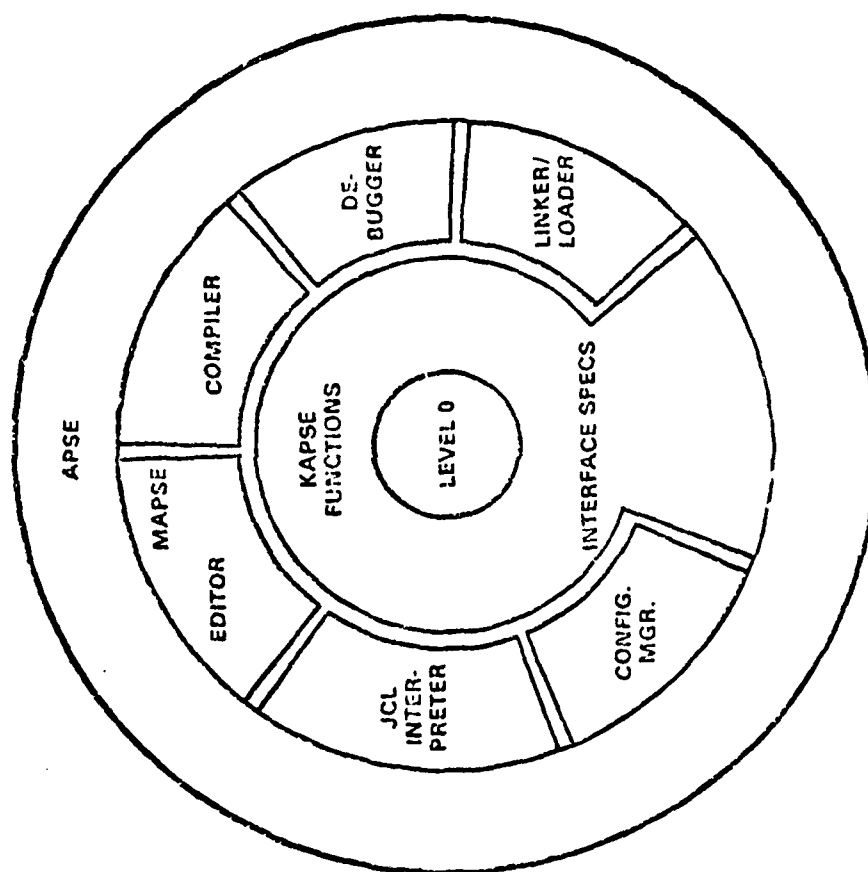
KAPSE INTERFACE TEAM INDUSTRY AND ACADEMIA - KITIA

- MEMBERSHIP SELECTED COMPETITIVELY
- MEMBERSHIP BELONGS TO ORGANIZATION
- VOLUNTEER EFFORT - ONE THIRD COMMITMENT REQUIRED
- CHAIRED BY DR. EDGAR SIBLEY
- ESTABLISHED FEBRUARY 1982 TO PROVIDE TECHNICAL SUPPORT TO THE KIT.

#



CSC



FUNDAMENTAL DEFINITIONS

- LEVEL 0: HARDWARE AND FOUNDATION HOST OPERATING SYSTEM
- LEVEL 1: KERNEL APSE (KAPSE)
 - PROVIDES DATA BASE, COMMUNICATION, RESOURCES, AND RUN-TIME SUPPORT FOR EXECUTING ADA PROGRAMS
 - PROVIDES A VIRTUAL HOST-INDEPENDENT PORTABLE INTERFACE
- LEVEL 2: MINIMAL APSE (MAPSE)
 - MINIMAL, NECESSARY, AND SUFFICIENT TOOL SET
- LEVEL 3: ADA PROGRAMMING SUPPORT ENVIRONMENT (APSE)
 - COMPREHENSIVE TOOL SET, MAPSE PLUS EXTENSIONS

DEFINITIONS

INTEROPERABILITY: INTEROPERABILITY IS THE ABILITY OF APSES TO EXCHANGE DATA BASE OBJECTS AND THEIR RELATIONSHIPS IN FORMS USABLE BY TOOLS AND USER PROGRAMS WITHOUT CONVERSION. INTEROPERABILITY IS MEASURED IN THE DEGREE TO WHICH THIS EXCHANGE CAN BE ACCOMPLISHED WITHOUT CONVERSION.

TRANSPORTABILITY: TRANSPORTABILITY OF AN APSE TOOL IS THE ABILITY OF THE TOOL TO BE INSTALLED ON A DIFFERENT KAPSE; THE TOOL MUST PERFORM WITH THE SAME FUNCTIONALITY IN BOTH APSES. TRANSPORTABILITY IS MEASURED IN THE DEGREE TO WHICH THIS INSTALLATION CAN BE ACCOMPLISHED WITHOUT REPROGRAMMING. PORTABILITY AND TRANSFERABILITY ARE COMMONLY USED SYNONYMS.

DEFINITIONS (CONTINUED)

REUSABILITY:

REUSABILITY IS THE ABILITY OF A PROGRAM UNIT TO BE EMPLOYED IN THE DESIGN, DOCUMENTATION, AND CONSTRUCTION OF NEW PROGRAMS. REUSABILITY IS MEASURED IN THE DEGREE TO WHICH THIS REUSE CAN BE ACCOMPLISHED WITHOUT REPROGRAMMING.

REHOSTABILITY:

REHOSTABILITY OF AN APSE IS THE ABILITY OF THE APSE TO BE INSTALLED ON A DIFFERENT HOST. REHOSTABILITY IS MEASURED IN THE DEGREE TO WHICH THE INSTALLATION CAN BE ACCOMPLISHED WITH NEEDED RE-PROGRAMMING LOCALIZED TO THE KAPSE. ASSESSMENT OF REHOSTABILITY INCLUDES ANY NEEDED CHANGES TO NON-KAPSE COMPONENTS OF THE APSE, IN ADDITION TO THE CHANGES TO THE KAPSE.

RETARGETABILITY:

RETARGETABILITY IS THE ABILITY OF A TARGET-SENSITIVE APSE TOOL TO ACCOMPLISH THE SAME FUNCTION WITH RESPECT TO ANOTHER TARGET. RETARGETABILITY IS MEASURED IN THE DEGREE TO WHICH THIS CAN BE ACCOMPLISHED WITHOUT MODIFYING THE TOOL. NOT ALL TOOLS WILL HAVE TARGET SPECIFIC FUNCTIONONS.

DEFINITIONS (CONTINUED)

HOST: A HOST IS A COMPUTER SYSTEM UPON WHICH AN APSE RESIDES, EXECUTES AND SUPPORTS Ada SOFTWARE DEVELOPMENT AND MAINTENANCE. EXAMPLES ARE IBM 360/370, VAX 11/730, CDC 6000/7000, DEC 20 AND UNIVAC 1110 WITH ANY OF THEIR RESPECTIVE OPERATING SYSTEMS.

TARGET: A TARGET IS A COMPUTER SYSTEM UPON WHICH Ada PROGRAMS EXECUTE.

REMARK: HOSTS ARE, IN FACT, ALSO TARGETS, IN/AS MUCH AS THE APSE IS WRITTEN IN Ada. A TARGET MIGHT NOT BE CAPABLE OF SUPPORTING AN APSE. AN EMBEDDED TARGET IS A TARGET WHICH IS USED IN MISSION CRITICAL APPLICATIONS. EXAMPLES OF EMBEDDED TARGET COMPUTER SYSTEMS ARE AN/AWK-14, AN/UYSK-43 AND COMPUTER SYSTEMS CONFORMING TO MIL-STD-1750A AND MIL-STD-1862 (NEBULA).

**OBJECTIVES OF THE KIT'S
APSE INTEROPERABILITY
AND TRANSPORTABILITY EFFORT**

- TO DEVELOP REQUIREMENTS FOR APSE IT.
- TO DEVELOP GUIDELINES, CONVENTIONS AND STANDARDS (GCS) TO BE USED TO ACHIEVE IT OF APSES.
- TO DEVELOP APSE TOOLS WHOSE INTEGRATION INTO BOTH THE AWE AND AHS WILL SERVE TO SURFACE AWE IT INTERFACES AND INTERFACE PROBLEMS.

KIT OBJECTIVES

(CONTINUED)

- TO MONITOR THE AIE AND ALS DEVELOPMENT EFFORTS WITH RESPECT TO APSE IT.
- TO PROVIDE INITIATIVE AND GIVE A FOCAL POINT WITH RESPECT TO APSE IT.
- TO DEVELOP AND IMPLEMENT PROCEDURES TO DETERMINE COMPLIANCE OF APCE DEVELOPMENT WITH APCE IT REQUIREMENTS, GUIDELINES, CONVENTIONS AND STANDARDS.

KIT SCHEDULE, MILESTONES, DELIVERABLES

1992

- THE KIT AND THE INDUSTRY/ACADEMIA TEAM GROUPS SELECTED AND WORKING
- DOCUMENTATION AND DESIGN REVIEWS OF AIE AND ALS APCE EFFORTS
- REQUIREMENTS DEVELOPED FOR THREE OR MORE APSE IT TOOLS
- CONTRACT LET FOR THREE OR MORE APCE IT TOOLS
- FIRST DRAFT OF APSE IT REQUIREMENTS
- SECOND DRAFT OF APCE IT REQUIREMENTS
- ROUGH DRAFT OF CONVENTIONS AND STANDARDS
- DRAFT PROCEDURES DETERMINATION OF AIE IT COMPLIANCE
- APSE IT CONFIGURATION MANAGEMENT PLAN
- TWO TECHNICAL REPORTS CONCERNING AIE IT EFFORTS

KIT SCHEDULE (CONTINUED)

1933

- **UPDATED APSE IT PLAN**
- **DOCUMENTATION, DESIGN AND TEST REVIEWS OF AIE AND ALS APSE EFFORTS**
- **TOOL DEVELOPMENT AND TOOL DEVELOPMENT REVIEWS**
- **THREE OR MORE APSE IT TOOLS DELIVERED AND INTEGRATED INTO THE AIE AND ALS APSES**
- **THIRD DRAFT OF APSE IT REQUIREMENTS**
- **SECOND DRAFT OF CONVENTIONS AND STANDARDS**
- **SECOND DRAFT OF PROCEDURES FOR DETERMINATION OF APSE IT COMPLIANCE**
- **FIRST DRAFT INTERFACE SPECIFICATIONS**
- **TWO TECHNICAL REPORTS CONCERNING APSE IT EFFORTS**

1934

SCHEDULE (CONTINUED)

1934

- UPDATED APSE IT PLAN
- DOCUMENT REVIEWS OF AIE AND ALS EFFORTS
- TESTING AND MODIFICATION OF THREE OR MORE APSE TOOLS COMPLETE
- FINAL DRAFT OF APSE REQUIREMENTS
- THIRD DRAFT OF CONVENTIONS AND STANDARDS
- FINAL DRAFT OF PROCEDURES FOR DETERMINATION OF APSE IT COMPLIANCE
- SECOND DRAFT INTERFACE SPECIFICATIONS
- TWO TECHNICAL REPORTS CONCERNING APSE IT EFFORTS

1935

- DOCUMENTATION, DESIGN AND TEST REVIEWS OF AIE AND ALS APSE EFFORTS
- FINAL DRAFT OF CONVENTIONS AND STANDARDS
- FINAL DRAFT OF INTERFACE SPECIFICATIONS
- TWO TECHNICAL REPORTS CONCERNING APSE IT EFFORTS

TEAM ORGANIZATION

- THE KIT AND KITIA ARE ORGANIZED ACCORDING TO THE
CATEGORIZATION OF THE MAPSE-KAPSE VIRTUAL INTERFACES:

GROUP 1, USER INTERFACES (WG.1)

GROUP 2, DATA INTERFACES (WG.2)

GROUP 3, RUN-TIME SERVICES INTERFACES (WG.3)

GROUP 4, MISCELLANEOUS (WG.4)

KAPSE INTERFACES

PROGRAM INVOCATION AND CONTROL

- INITIATION, SUSPENSION, RESUMPTION, TERMINATION
- SPECIFICATION OF PARAMETERS AND OPTIONS
- EXECUTION MONITORING
- REDIRECTION OF INPUT AND OUTPUT
- RETURN OF TERMINATION STATUS

AD-A141 576

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM PUBLIC REPORT VOLUME 3(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 25 OCT 83
NOSC/TD-552-VOL-3

575

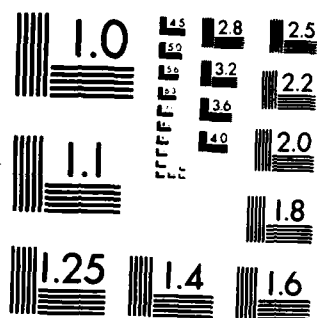
UNCLASSIFIED

F/G 9/2

NL



END
DATE
FILMED
7 84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

KAPSE INTERFACES

LOGON/LOGOFF SERVICES

- USER AUTHENTICATION
- ACCOUNTING
- SAVE/RESTORE WORKING ENVIRONMENT

KAPSE INTERFACE

DEVICE INTERACTIONS

- TRANSMISSION OF DATA AND CONTROL
- CHARACTER SET STANDARDIZATION
- COMMUNICATION PROTOCOLS
- STANDARDIZATION OF SUPPORTED DEVICES

KAPSE INTERFACES

DATA BASE SERVICES

- BASIC OBJECT OPERATIONS
- ATTRIBUTE MANIPULATION AND QUERY
- CATEGORIES
- VERSION CONTROL
- ACCESS CONTROL
- HISTORY MAINTENANCE
- PARTITIONS
- ARCHIVE AND BACK-UP

KAPSE INTERFACES

Ada RUN-TIME SYSTEM

- EXCEPTION HANDLING
- PROGRAM INITIALIZATION/TERMINATION
- TASKING
- STANDARD INPUT/OUTPUT

KAPSE INTERFACES

INTER-TOOL DATA INTERFACES

FOR

- **DOCUMENTATION TOOLS**
- **REQUIREMENTS AND SPECIFICATION TOOLS**
- **PROJECT MANAGEMENT TOOLS**
- **CONFIGURATION MANAGEMENT TOOLS**
- **SOFTWARE GENERATION TOOLS**

OF IMMEDIATE INTEREST

- **STANDARD PROGRAM LIBRARY FORMAT**
- **STANDARD OBJECT FILE FORMAT**

KAPSE INTERFACES

Ada INTERMEDIATE LANGUAGE (DI. IIA)

RETAINS Ada PROGRAM INFORMATION FOR:

- **COMPILER PHASES**
- **OPTIMIZERS**
- **STATIC AND DYNAMIC ANALYZERS**
- **DEBUGGERS**
- **TEST TOOLS**
- **PRETTY-PRINTERS**
- **SYNTAX-ORIENTED EDITORS**

KAPSE INTERFACES

GENERAL ISSUES

- PERFORMANCE MEASUREMENT
- RECOVERY MECHANISMS
- DISTRIBUTED APSES
- SECURITY
- TARGET SUPPORT



FIRST YEAR ACHIEVEMENTS

- TWO PUBLIC REPORTS PUBLISHED DOCUMENTING FIRST YEAR'S ACTIVITIES
- TWO APSE I & T TOOLS CONTRACTED
- SEVERAL TECHNICAL BRIEFINGS TO PROFESSIONAL GROUPS (ADATEC, DPMA) AND MEMBER ORGANIZATIONS
- PRELIMINARY DRAFT OF I & T REQUIREMENTS
- PRELIMINARY DRAFT OF A STANDARD INTERFACE SET

- TWO APSE TOOLS UNDER DEVELOPMENT FOR THE AIE AND ALS:
 - CONFIGURATION MANAGEMENT SYSTEM (CONFIGURE),
IMPLEMENTED BY CSC TO AUTOMATICALLY MAINTAIN
THE CONSISTENCY AMONG RELATED DATA BASE OBJECTS
 - ADA INTERACTIVE MONITOR (AIM),
IMPLEMENTED BY TI TO SHARE ONE PHYSICAL TERMINAL
AMONG MANY VIRTUAL TERMINALS

CSC

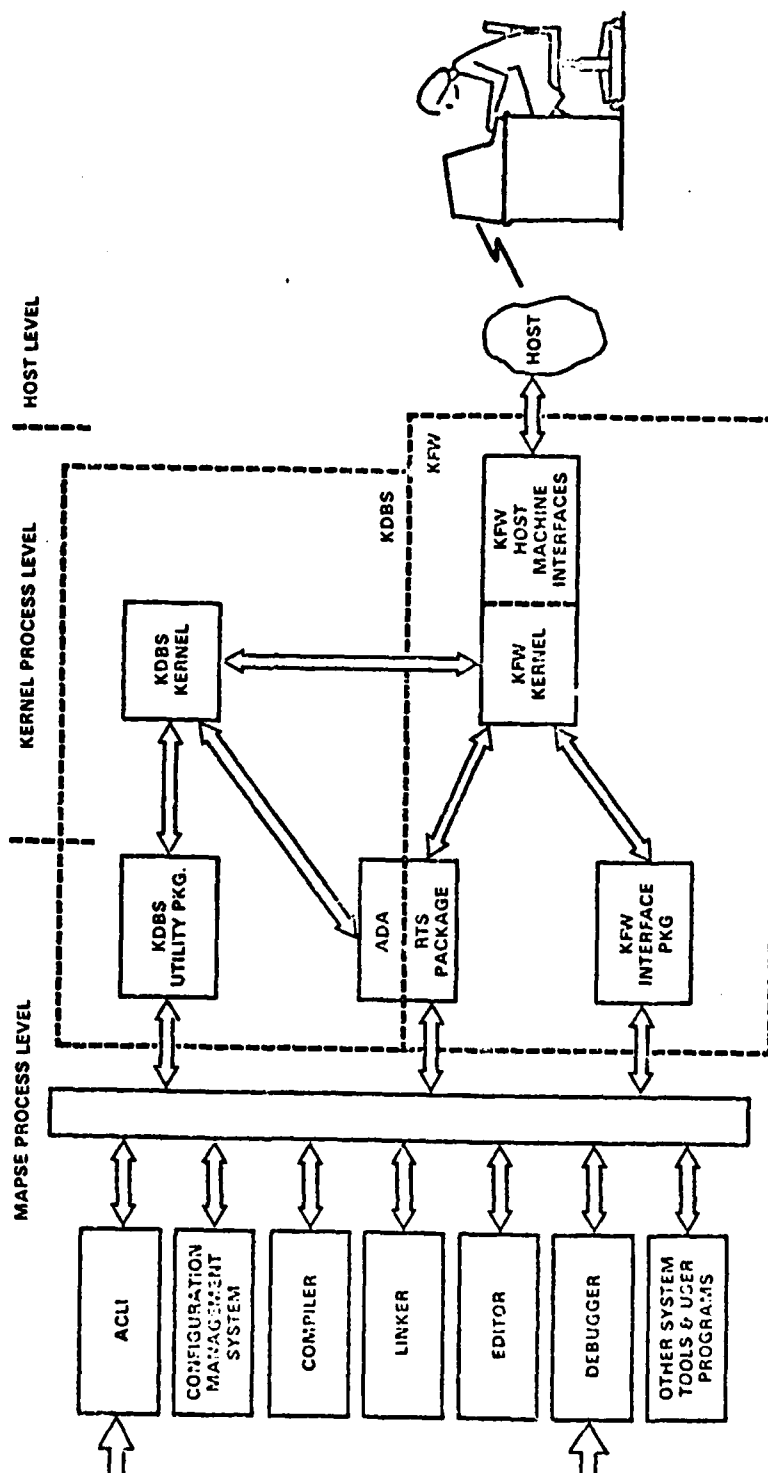
IMPORTANT TEAM DOCUMENTS

- DOCUMENTS CURRENTLY UNDERGOING INTENSIVE EVALUATION
AND REVIEW BY THE KIT AND KITIA:
 - STONEMAN II, AN ELABORATION OF STONEMAN INTERFACES
(JANUARY 1983)
 - KIT STRATEGY STATEMENT (JANUARY 1983)
 - INTEROPERABILITY AND TRANSPORTABILITY REQUIREMENTS
WORKING PAPER (FEBRUARY 1983)
 - ADA PACKAGE SPECIFICATIONS FOR STANDARD INTERFACE
SET (FEBRUARY 1983)

DIFFERENT INTERFACE PERIMETERS

- THE ORIGINAL NOTION OF THE KAPSE INTERFACE MAY NOT ACHIEVE
I & T OF TOOLS, THEREFORE, DIFFERENT INTERFACE PERIMETERS
HAVE BEEN PROPOSED:
 - STANDARD INTERFACE SET (SIS)
 - SUBSET MAPSE (SMAPSE)
 - REQUIRED ADA HOST DEVELOPMENT SYSTEM (RAHDS)
 - BOOTSTRAP APSE STANDARD INTERFACE SET (BASIS)

DIFFERENT I & T INTERFACE PERIMETERS



- THE FOLLOWING RULES APPEAR TO SUMMARIZE THE INTENT OF THE DIFFERENT INTERFACE PERIMETERS:
 - I & T REQUIREMENTS OF A MAPSE ARE SATISFIED BY A CONFORMING KAPSE IMPLEMENTATION
 - I & T REQUIREMENTS OF AN APSE ARE SATISFIED BY A CONFORMING MAPSE IMPLEMENTATION

CSC

IMMEDIATE EMPHASIS

- IMPROVE KIT AND KITIA COORDINATION THROUGH MORE FREQUENT
TECHNICAL EXCHANGES
- REFINE I & T REQUIREMENTS DOCUMENT
- DEVELOP PRELIMINARY STANDARD INTERFACE SET FOR PUBLIC
REVIEW
- REVIEW REVISED AIE B5 SPECIFICATIONS

- DEVELOP A PRELIMINARY DRAFT OF A STANDARD INTERFACE SET FOR PUBLIC REVIEW DURING 4TH QUARTER 1983
- NEW GROUP (WG.O) CHARTERED TO PROVIDE MAJOR TECHNICAL CONTRIBUTIONS DURING NEXT 4 MONTHS
- THE EXISTING WORKING PAPER FOR A STANDARD INTERFACE SET WAS SELECTED AS THE FOUNDATION DOCUMENT TO GUIDE THE EFFORT

- FORMAL SPECIFICATIONS OF KAPSE INTERFACE SEMANTICS
- KAPSE INTERFACE VALIDATION
- INTERFACE EXTENSIBILITY AND EXPANDABILITY
- COMMAND LANGUAGE AND DEBUGGER INTERFACES AND REQUIREMENTS
- BOOTSTRAP APSE STANDARD INTERFACE SET-BASIS
- CLASSES OF DISTRIBUTED APSES
- INTERFACES FOR TARGETS - KARTSE, TASK, RTOS...

- NEED TO DEVELOP GUIDELINES, CONVENTIONS, AND STANDARDS HAS INCREASED BECAUSE OF:
 - AVAILABILITY OF COMMERCIAL ENVIRONMENTS, ROLM, TELESOFT,...
 - MORE AMBITIOUS ENVIRONMENTS EXPECTED, ALS/NAVY
- SOME OBJECTIVES MUST BE RESOLVED CONCURRENTLY:
 - DEVELOPMENT OF PRELIMINARY GCSS FOR NEAR TERM PLATFORM
 - DEVELOPMENT OF LONG TERM I & T REQUIREMENTS
- PRODUCTIVITY HAS BEEN IMPACTED BY:
 - DIFFICULTY IN FOCUSING UPON REAL OBJECTIVES
 - DIVERSITY OF TEAMS' BACKGROUND AND EXPERIENCE
 - DEDICATED EFFORT ESSENTIAL TO ACHIEVE STABLE PLATFORM RAPIDLY.

CSC

AVAILABLE DOCUMENTS

- KIT PUBLIC REPORT VOLUME 1 - NOSC TD 509, APRIL 1982
 - AVAILABLE THROUGH NTIS, \$19.50
 - ORDER No. AD A115 590
- KIT PUBLIC REPORT VOLUME 2 - NOSC TD 552, OCTOBER 1982
 - AVAILABLE THROUGH NTIS (MAY), \$45.00
 - ORDER No. AD A123 136